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

MATHEUS DE MELLO FREIRE

# Extração e Verificação de Modelos para Sistemas em Evolução

Trabalho de Graduação.

Prof. Dr. Lucio Mauro Duarte

Porto Alegre, dezembro de 2012.

# Agradecimentos

Gostaria de agradecer ao meu orientador, pelo suporte e conselhos durante o acompanhamento deste trabalho. Agradeço à minha família e amigos, pelo apoio dado. E por último, à minha namorada, a qual esteve ao meu lado durante todo este tempo, dando seu apoio para que este trabalho fosse realizado.

# 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 ..... 11**
**1.3 Objetivo ..... 12**
**1.4 Organização do Texto ..... 12**
**2 REFERENCIAL TEÓRICO ..... 14**
**2.1 Modelos ..... 14**
2.1.1 Máquinas de Estados Finitos ..... 15
2.1.2 Labelled Transition Systems ..... 15
**2.2 Extração e Verificação de Modelos ..... 17**
2.2.1 Ferramentas ..... 17
2.2.1.1 LTSA ..... 17
2.2.1.2 LTSE ..... 21
**3 METODOLOGIA ..... 22**
**3.1 Alteração Manual de Modelo ..... 22**
**3.2 Levantamento de Casos de Teste ..... 23**
**3.3 Instrumentação do Código ..... 23**
**3.4 Geração de Trace Logs ..... 25**
**3.5 Extração de Modelo de Comportamento ..... 26**
**3.6 Verificação de Modelos ..... 26**
**4 EXPERIMENTOS ..... 28**
**4.1 Traffic Lights ..... 28**
**4.2 Editor ..... 49**
**5 CONCLUSÃO ..... 65**
**REFERÊNCIAS ..... 66**

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| LTS | Labelled Transition System |
| LTSA | Labelled Transition System Analyser |
| LTSE | Labelled Transition System Extractor |

# LISTA DE FIGURAS

Figura 2.1: Visualização gráfica de um modelo e sua representação textual. ..... 18
Figura 2.2: Execução de um possível passeio do modelo. ..... 18
Figura 2.3: Modelo definido como propriedade e novo modelo a ser comparado ..... 19
Figura 2.4: Modelo definido como propriedade com o estado de erro. ..... 19
Figura 2.5: Modelo a ser comparado com a propriedade ..... 20
Figura 2.6: Modelo resultante da verificação ..... 20
Figura 4.1: Modelo 1 - Traffic Lights com as cores Red e Green, extraído do código do sistema ..... 31
Figura 4.2: Validação das transições entre estados permitida pela ferramenta LTSA. .. 32
Figura 4.3: Modelo 2 - Traffic Lights com as cores Red, Yellow e Green, alterado manualmente ..... 33
Figura 4.4: Modelo 3 - Traffic Lights com as cores Red, Yellow e Green, extraído do código alterado do sistema. ..... 35
Figura 4.5: Modelo definido como propriedade. ..... 37
Figura 4.6: Verificação dos modelos 2 e 3 ..... 38
Figura 4.7 : Modelo 4 – Luz intermitente - Modelo gerado manualmente ..... 39
Figura 4.8: Modelo 5 – Luz Intermitente Modelo gerado automaticamente, implementação sem exceção ..... 42
Figura 4.9: Verificação dos modelos 4 e 5. ..... 43
Figura 4.10: Modelo 6 - Luz Intermitente - Modelo gerado automaticamente, implementação utilizando exceção ..... 46
Figura 4.11 Verificação dos modelos 4 e 6 ..... 47
Figura 4.12: Modelo 1: Editor com propriedades e operações originais, extraído do código do sistema ..... .53
Figura 4.13: Modelo 2: Operação saveas adicionada manualmente, sem alterar as propriedades ..... 54
Figura 4.14: Modelo 3: Operação saveas adicionada no código. Modelo gerado de forma automática, sem alterar as propriedades ..... 55
Figura 4.15 – Verificação entre os modelos 2 e 3 ..... 56
Figura 4.16: Modelo 4, modificado manualmente com Undo/Redo ..... 58
Figura 4.17: Modelo 5, extraído do código-fonte com Undo/Redo ..... 61
Figura 4.18: Verificação dos modelos 4 e 5 ..... 62

# LISTA DE TABELAS

Tabela 4.1: Descrição dos estados e arcos do modelo 1 ..... 31
Tabela 4.2: Resultados dos experimentos realizados com Traffic Lights ..... 48
Tabela 4.3: Descrição dos estados e arcos do modelo 1 ..... 53
Tabela 4.4: Resultados dos experimentos realizados com Editor ..... 63

# RESUMO

Softwares estão sempre em constante evolução e atualização. Durante o ciclo de vida de um sistema, muitas modificações ocorrem, em relação às funcionalidades definidas em sua primeira especificação. É desejável que durante todo o ciclo de vida do sistema, o mesmo execute corretamente as funções para as quais foi projetado, mesmo na ocorrência de modificações. Para isto, o uso de verificação de modelos e teste de software tem desempenhado uma importante função na busca por melhor qualidade de software.

Este trabalho busca utilizar técnicas de teste de software, verificação e extração de modelos para obter modelos os quais representem os comportamentos corretos de sistemas mesmo após modificações. O objetivo do trabalho é estabelecer um processo de validação de sistemas em relação a propriedades estabelecidas o qual permita não apenas verificar se as propriedades são preservadas em um modelo do sistema, mas também se o comportamento da implementação gerada corresponde ao definido no modelo.

Utilizando uma modelagem manual a partir de um modelo pré-existente, é definida uma única propriedade, onde os comportamentos esperados do sistema em questão são englobados por esta propriedade. Após a modificação do código, um novo modelo é gerado com base no código do sistema e verificado contra a propriedade estabelecida anteriormente, a fim de validar se os comportamentos encontrados no código estão de acordo com a especificação do sistema.

Validou-se o processo utilizando dois sistemas previamente codificados, onde um conjunto de alterações foi definido para cada um deles. Através do teste de software e extração de modelos, construiu-se um modelo com comportamentos que são encontrados na implementação de cada sistema. Com a verificação de modelos, a validade destes comportamentos foi comprovada ao realizar tal verificação contra um modelo o qual contém todos os comportamentos de um modelo cuja correção é previamente conhecida, além dos comportamentos adicionados pelas modificações.

O processo proposto facilita o suporte para detecção de erros tanto nos modelos extraídos do código quanto nos modelos modificados manualmente, além de apresentar um modelo sem erros quando os comportamentos são todos corretos.

**Palavras-Chave:** Teste de Software, Verificação de Modelos, Extração de Modelos de Comportamento, Evolução de Sistemas.

**Model Checking and Model Extraction during software**

# ABSTRACT

Software is always evolving and changing. During a software lifecycle, many updates are done regarding functions defined on early specification. It's desirable that during all system lifecycle, all functions are correctly executed, even after software updates. In order to get that, using model checking and software testing has been useful to reach better software quality.

This paper uses software testing, model checking and behavior model extraction techniques to obtain models to represent valid software behavior even after an update is done. The purpose of this paper is to get a system validation process against system properties where it's possible to verify if a mode still represents system properties and to verify if the model behavior is the same as the behavior found on system implementation.

Using manual modeling, only one property is defined, there expected system behavior are represented on this single property. After code update, a new model is generated based on the code and it is verified against the property, in order to verify if the code matches the system specification.

Two system codes were used on this verification, where a set of updates were defined for each one of them. Using software testing and model extraction, a model was built with behavior found on each system. With model checking, behavior validation was done using this built model against a previous correct known model.

This process makes bug detection easier on both extracted models as manually updated models. It also results on a model with no errors when all behaviors found are correct.

**Keywords:** Software Testing, Model Checking, Behavior Model Extraction, Software Evolution.

# 1 INTRODUÇÃO

## 1.1 Abordagem do Tema

Para garantir um nível adequado de qualidade de um software, técnicas como Teste de Software (PEZZÈ, 2008) e Verificação de Modelos (CLARKE, 1999) são comumente utilizadas, aumentando o grau de confiabilidade do sistema em questão. Um bom conjunto de testes aumenta a cobertura de comportamentos que podem ser observados em um sistema, pois possibilita que um maior número de erros possa ser encontrado. Testes de software podem ser utilizados para avaliar o grau de adequação do sistema em relação às necessidades reais dos usuários, onde o objetivo a ser alcançado é a utilidade do software. Outro objetivo, a confiabilidade, pode ser alcançado através de testes pela checagem de uma implementação em relação à especificação definida.

Um conjunto de testes dificilmente tem o objetivo de encontrar todos os erros em um sistema, mas sim de encontrar o maior número possível de problemas em qualquer artefato, como código, especificação e testes. Utilizam-se testes para que os graus de confiabilidade e satisfação descritos anteriormente atendam às necessidades dos usuários. Com isso, o uso de teste de software ajuda na procura de problemas simples em pequenas unidades do sistema e em erros gerados pela integração de uma ou mais porções do código. Seu uso recorrente tem como objetivo validar que após correções e modificações na implementação, os requisitos do sistema sejam respeitados.

Porém, um conjunto de testes não tem como objetivo testar todas as possibilidades de uma função. Por exemplo, ao testar uma funcionalidade que necessita de números inteiros como valor de entrada, testam-se valores positivos, negativos, zero, próximos aos limites do conjunto de inteiros estabelecido pela arquitetura do hardware, com o objetivo de testar diferentes classes de comportamento do sistema. É difícil analisar o sistema como um todo utilizando testes que individualmente se preocupam em validar comportamentos pontuais no sistema, por isso vemos valor em utilizar outras técnicas em conjunto a testes para melhorar a confiabilidade dos comportamentos do software em relação à especificação.

Verificação de modelos pode ser definida como um processo automatizado onde um conjunto de propriedades desejáveis do sistema é verificado contra um modelo que o representa. O objetivo da verificação é observar se as propriedades são satisfeitas pelo modelo. Por ser um processo automatizado, utilizam-se ferramentas denominadas verificadores, onde ao final, obtêm-se dois possíveis resultados: especificação totalmente satisfeita ou violação de uma ou mais propriedades. Visualmente, o verificador pode apresentar as violações encontradas na forma de um modelo de contraexemplo, o qual consiste de transições onde as violações das propriedades podem ser observadas. Caso a especificação seja respeitada, o modelo não apresentará tais

transições e será considerado, para este trabalho, correto. Utilizaremos o termo correção para definir um modelo ou implementação que está de acordo com a especificação e propriedades definidas.

Caso o resultado da verificação não apresente violações das propriedades, não necessariamente signifique que todos os comportamentos do sistema são corretos, pois modelos são abstrações que podem não possuir todos os comportamentos possíveis. A verificação sem erros apenas conclui que para os comportamentos incluídos no modelo, nenhuma violação da especificação ocorre. O grande problema em reunir o maior número de comportamentos relevantes no modelo é a sua construção, o que pode ser trabalhoso de forma manual e nem sempre representar os comportamentos reais do sistema. Por isso, utilizar testes sobre o sistema real pode trazer uma importante contribuição para a verificação de modelos.

Dada à preferência por modelos com comportamentos reais, é interessante obter-se um modelo a partir da implementação, contendo apenas os comportamentos da mesma. Para isso, pode-se utilizar extração de modelos de comportamento, onde definido um conjunto de testes a ser executado sobre a implementação do sistema, gera-se um modelo que possui uma cobertura de comportamentos reais tão boa quanto a cobertura dos testes utilizados.

Ao juntar teste de software com a verificação de modelos, o modelo resultante conterá apenas comportamentos reais validados contra um conjunto de propriedades a serem respeitadas. A abordagem de extração definida em (DUARTE, 2006) utiliza *traces* reais do sistema para gerar um modelo que o represente. Tal abordagem, baseada no código já existente, faz uso da combinação de informações estáticas e dinâmicas, gerando modelos corretos em relação à implementação e testes executados, ou seja, erros encontrados no modelo são erros existentes no código. No entanto, dependendo do conjunto de *traces* gerados, alguns comportamentos executáveis podem não estar no modelo.

## 1.2 Motivação

A abordagem discutida acima tem como escopo apenas realizar a extração e posterior verificação de modelos de comportamento em sistemas já finalizados. No entanto, durante o ciclo de vida de um sistema, é comum que novas funcionalidades sejam inseridas após existir uma versão de código funcional e estável. Tais modificações levam a alterações na especificação do sistema e causam impacto na validade do modelo e do código existentes. Deste modo, faz-se necessário um processo pelo qual novas propriedades incluídas na especificação possam gerar modelos atualizados. Tal processo deve assegurar que comportamentos pré-existentes continuem sendo respeitados, assim como os novos comportamentos gerados pelas propriedades incluídas.

Tendo um modelo finalizado, é simples modifica-lo manualmente para acrescentar uma nova funcionalidade, dada a facilidade de visualiza-lo em sua totalidade. Modificar o código pode gerar efeitos colaterais em outras porções de código, sem que o programador tenha conhecimento antes de realizar um conjunto de testes que venha a detectar estes erros. Assim, parte-se da hipótese de que a modificação manual do modelo já existente é um processo simples, mais fácil de realizar se comparado à modificação e criação de testes e código fonte. Dada à natureza simples das

modificações propostas, ao validar que os comportamentos gerados pelas modificações são válidos, considera-se que os modelos modificados manualmente refletem o comportamento esperado do sistema tanto quanto o modelo inicial estável. Logo, tais modelos serão utilizados como propriedade durante a verificação de modelos extraídos do código fonte. Podemos partir deste princípio, já que a abordagem definida em (DUARTE, 2006) gera modelos cujos comportamentos sempre são encontrados na implementação. Apesar desta característica, não é garantida a completude do modelo, já que o conjunto de testes utilizados na extração não precisa necessariamente cobrir todos os comportamentos do sistema, por isso, não faz parte do escopo deste trabalho avaliar a completude dos modelos gerados.

Ao propor uma alteração na especificação, três artefatos serão potencialmente modificados: modelo, conjunto de testes e código. A modificação de qualquer um destes artefatos está sujeita a erros, os quais serão analisados durante os experimentos realizados neste trabalho.

## 1.3 Objetivo

Com este trabalho, visa-se estudar o processo de teste e verificação de modelos encontrados em (DUARTE, 2006) quando utilizado em sistemas não finalizados, ou seja, sistemas que sofrem alterações após já possuírem ao menos um protótipo funcional.

Serão utilizadas apenas alterações as quais incluem novas funcionalidades, não fazendo parte do escopo deste trabalho alterações que apenas modifiquem comportamentos já existentes ou os excluam. O resultado esperado é que os modelos gerados sejam corretos para a maior variedade de alterações utilizadas e que o uso de modelos manuais ajudem a encontrar possíveis erros de implementação nos sistemas modificados.

Não faz parte deste trabalho utilizar um modelo para gerar código atualizado, visto que o código gerado a partir de uma estrutura abstrata como um modelo, seria muito complexo.

## 1.4 Organização do texto

Este trabalho está estruturado da seguinte maneira:

No segundo capítulo, é descrito um referencial teórico, onde se apresentam os conceitos utilizados para a realização dos experimentos.

No terceiro capítulo, é apresentada a metodologia utilizada para todos os experimentos. Todos os passos comuns aos experimentos realizados são descritos nesta etapa.

No quarto capítulo, cada sistema e alterações propostas para os experimentos são apresentados, assim como os resultados obtidos e uma discussão individual de cada experimento.

No quinto e último capítulo, apresentamos a conclusão do trabalho, além de discutir melhorias as quais podem ser incorporadas em trabalhos futuros.

# 2 REFERENCIAL TEÓRICO

Este capítulo descreve diferentes tipos de modelos, incluindo o escolhido para os experimentos realizados. Também são apresentadas as ferramentas utilizadas para a geração dos modelos.

## 2.1 Modelos

Um modelo é uma representação do comportamento de um sistema, preservando atributos importantes do artefato real. São utilizados para representar requisitos, incorporando estruturas e informações sobre erros que podem ajudar na especificação de casos de teste.

Para uma melhor análise dos comportamentos descritos em um modelo, algumas características são desejáveis, tais quais:

- Legibilidade: Um modelo deve ser representado de tal forma que seja facilmente compreendido por quem o interprete. Todos os comportamentos modelados devem ser entendidos de uma maneira apenas, sem ambiguidades.
- Comportamentos Reais: Um modelo deve possuir apenas comportamentos que devem ser encontrados na implementação do sistema (ou encontrados, caso trate-se de um modelo extraído diretamente do código previamente implementado). Todo e qualquer comportamento que não deve fazer parte do escopo do modelo não deve ser representado no mesmo.
- Cobertura de Propriedades: Um modelo deve possuir comportamentos que englobem todas as propriedades definidas para o sistema representado.

O uso de modelos traz diversas vantagens, como encontrar falhas no processo antes mesmo de codificar o sistema, definir uma base para um plano de testes, reutilizar o modelo para versões futuras ou sistemas semelhantes ao modelado, compartilhar informações relevantes ao comportamento do sistema e permitir atualização incremental das características do sistema sem necessidade de retrabalho. Pode-se representar um modelo de diferentes maneiras, sejam elas formais ou não. Podemos citar como exemplo, máquina de estados finitos, gramáticas, grafos de fluxo de controle e diagramas de classes e objetos.

### 2.1.1 Máquina de Estados Finitos

Muitos sistemas podem ser interpretados como um conjunto finito de “estados”, os quais “memorizam” a porção relevante da execução do sistema até o momento, e um conjunto de possíveis ações que possibilitam a transição entre os diversos estados. Com um número finito de estados, o histórico inteiro de execução em geral não é memorizado, assim o sistema deve ser projetado de tal maneira a guardar apenas informações relevantes e “esquecer” o que não é. Assim, é possível implementar o sistema com um conjunto fixo de recursos, como um circuito implementado em hardware ou um software capaz de tomar decisões a partir de um conjunto limitado de dados ou a posição no código.

Tais sistemas, juntamente com analisadores léxicos e sistemas de busca de padrões (palavras, frases) podem facilmente ser representados na forma de uma Máquina de Estados Finitos (FSM, Finite State Machine). Formalmente, uma FSM é definida em (HOPCROFT, 2002) pela quíntupla $M = (Q, \Sigma, \delta, q0, F)$, tal que:

- Q é um conjunto finito não vazio de estados da FSM;
- $\Sigma$ é um conjunto de símbolos, denominado alfabeto de entrada da FSM;
- $\delta : Q \times \Sigma \rightarrow Q$ é a função de transição de estados da FSM e seu papel é o de indicar as transições possíveis em cada configuração da FSM. Esta função fornece para cada par "estado e símbolo de entrada" um novo estado para onde a FSM deverá mover-se.
- $q0 \in Q$ é denominado estado inicial da FSM finito. É o estado para o qual o reconhecedor deve ser levado antes de iniciar suas atividades.
- $F \subseteq Q$ é um subconjunto do conjunto Q dos estados da FSM, e contém todos os estados de aceitação ou estados finais da FSM finito. Estes estados são aqueles em que a FSM deve terminar o reconhecimento das cadeias de entrada que pertencem à linguagem que a FSM define. Nenhuma outra cadeia deve ser capaz de levar a FSM a qualquer destes estados.

Dada uma sequência de símbolos de entrada válidos do alfabeto $\Sigma$, representados por a1a2...an , o estado inicial q0, e uma função de transição $\delta(q0,a1) = q1$, o processamento do primeiro símbolo de entrada resultará na transição do estado q0 para o estado q1. Para cada símbolo de entrada remanescente, consulta-se a função de transição correspondente ao estado atual e o símbolo processado, resultando em um novo estado atingido pela execução, até que o último símbolo seja processado. Quando toda a sequência de símbolos for processada, verifica-se se o estado atingido pertence ao conjunto de estados finais F. Caso positivo, a entrada é aceita pela FSM, caso contrário ela é rejeitada.

### 2.1.2 Labelled Transition Systems

Uma máquina de estados finitos com apenas um estado inicial, nenhum estado final e com rótulos nas transições é um Labelled Transition System. Tal modelo é útil para

representar comportamentos, visto que as transições do LTS podem representar os métodos executados em um sistema.

Um LTS finito P, como descrito em (MAGEE, 1999) é uma quádrupla <S, A, Δ, q> onde:

- S ⊆ Estados é um conjunto finito de estados
- A = αP ∪ {τ}, onde αP ⊆ L denota o alfabeto de P
- Δ ⊆ S - {π} x A x S, denota uma relação de transição que mapeia de um estado e uma ação para outro estado.
- q ∈ S indica o estado inicial de P

Segundo (DUARTE, 2006), LTS têm sido utilizados com sucesso para modelar e analisar comportamentos de sistemas, mesmo complexos. Em conjunto com certas ferramentas para visualização e extração de modelos às quais utilizam LTS, esta representação é utilizada neste trabalho. Como os arcos de um LTS podem representar inputs do sistema, bem como métodos executados, é possível interpretar passeios dentro do LTS como possíveis execuções do sistema o qual o mesmo representa. Ao analisar tais passeios, é possível analisar os comportamentos contidos no modelo.

## 2.2 Extração e Verificação de Modelos

Ao utilizar verificação de modelos, é preciso definir propriedades com as quais o modelo deve ser validado. É possível definir propriedades como pequenas porções de um modelo ou como um modelo completo, representando todos os comportamentos válidos do sistema. Tais propriedades devem descrever os comportamentos esperados do sistema, de tal maneira que erros sejam detectados após a verificação, caso o modelo seja inválido de acordo com uma ou mais propriedades.

Construir um modelo a ser utilizado em um verificador é uma tarefa difícil de ser realizada manualmente, por isso pode-se utilizar um processo automatizado de extração de modelos (HOLZMANN, 1999). Tal processo tem por objetivo recolher informações do código do sistema para gerar um modelo o qual represente os comportamentos encontrados no sistema, sejam eles válidos ou inválidos.

Para realizar estas etapas, o uso de ferramentas que automatizem alguns processos faz-se necessário. Tais ferramentas são descritas a seguir.

### 2.2.1 Ferramentas

Para este trabalho, serão utilizadas as ferramentas LTSE e LTSA, para extração e verificação de modelos, respectivamente. Outras ferramentas, tais quais Spin, NuSMVou DiVinE Tool poderiam ser utilizadas.

#### *2.2.1.1 LTSA*

O *Labelled Transition System Analyser* (LTSA) é uma ferramenta utilizada para verificação de modelos descritos em LTS. A ferramenta pode ser utilizada para descrever um modelo textualmente a partir de um novo arquivo ou carregar um arquivo já existente com um modelo. Após compilar o arquivo e minimizar o número de estados e transições para obter um modelo mais legível e compacto, o usuário pode visualizar o modelo em sua representação gráfica como é mostrado na figura 2.1, ou simular uma execução do modelo para gerar possíveis passeios pelo mesmo, exemplificado na figura 2.2. Durante a execução, o LTSA percorre os estados do modelo levando em consideração os arcos escolhidos pelo usuário. A ferramenta irá permitir que apenas arcos válidos possam ser escolhidos, isto é, arcos cuja origem é o estado no qual a execução se encontra.

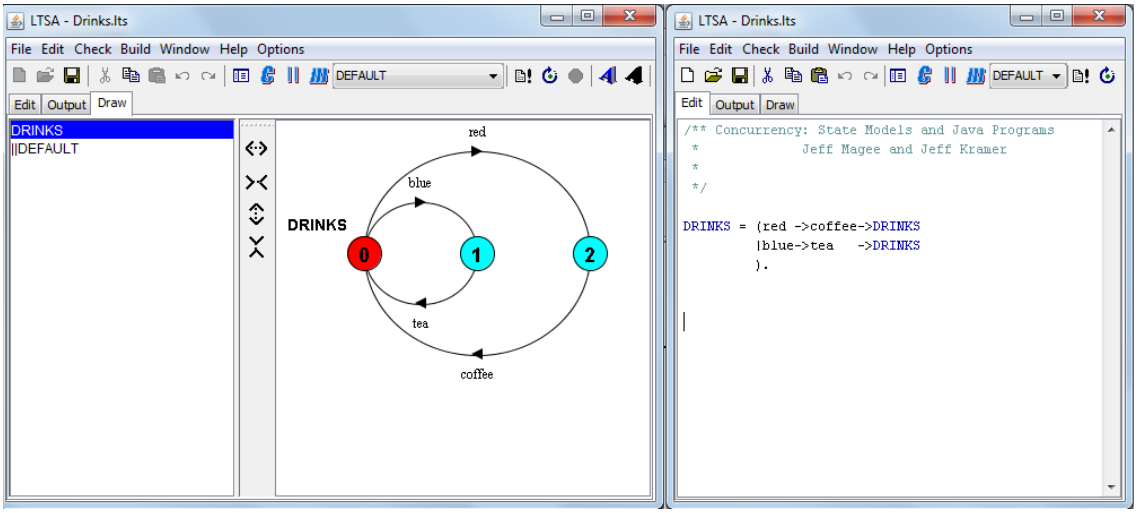


Figura 2.1: Visualização gráfica de um modelo (esquerda) e sua representação textual (direita).

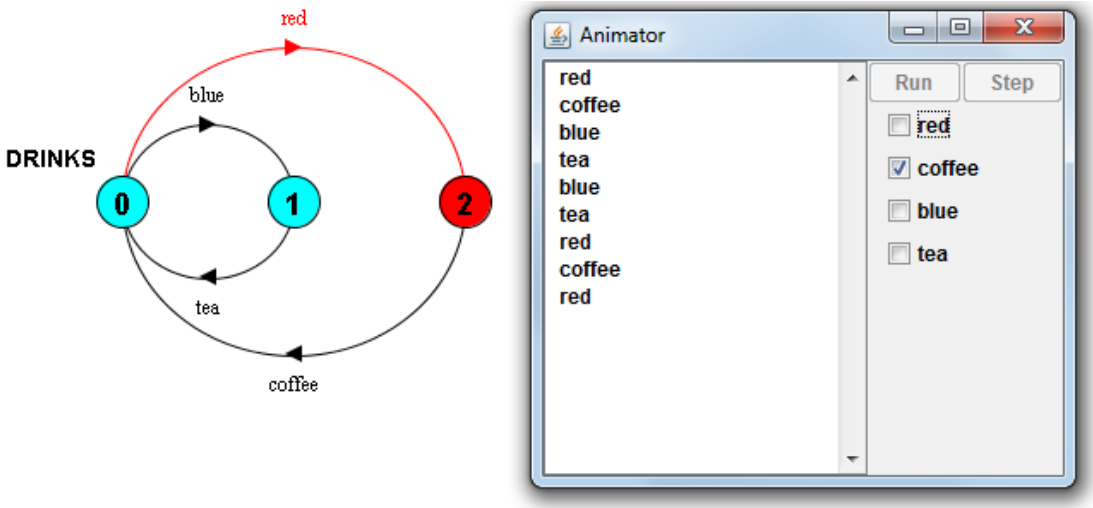


Figura 2.2: Execução de um possível passeio do modelo.

Além destas funcionalidades, é possível definir um modelo como uma propriedade. Para isto, basta utilizar o comando *property*, como é visto na figura 2.3. Isto faz com que o LTSA considere um modelo inteiro como a especificação do sistema, o qual contém todos os comportamentos considerados corretos. Quando um modelo é definido como propriedade, o LTSA cria um estado de erro nomeado -1, o qual será destino de todos os arcos os quais representam comportamentos não existentes no modelo levando em consideração o conjunto de arcos existentes. A presença deste estado de erro no modelo resultante da verificação indicará comportamentos não validos, enquanto sua ausência indicará que todos os comportamentos definidos pela propriedade são respeitados no novo modelo. Um exemplo disto é observado na figura 2.4.

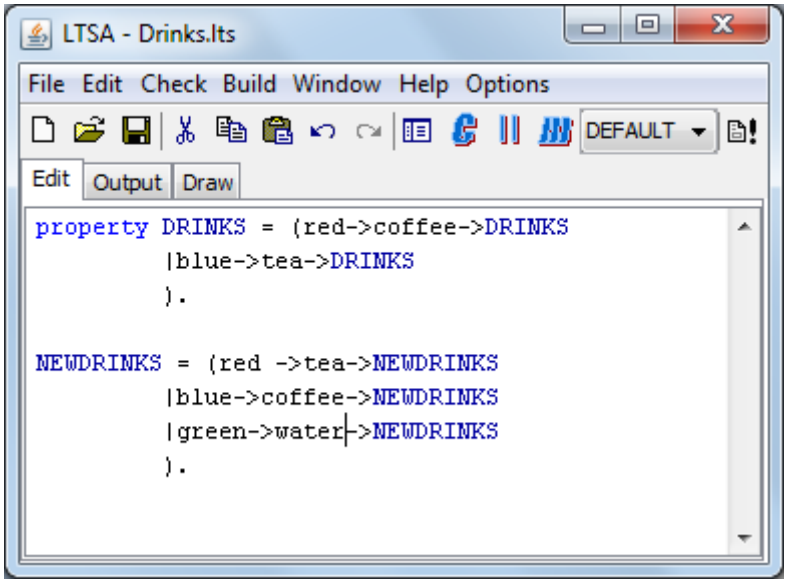


Figura 2.3: Modelo definido como propriedade e novo modelo a ser comparado.

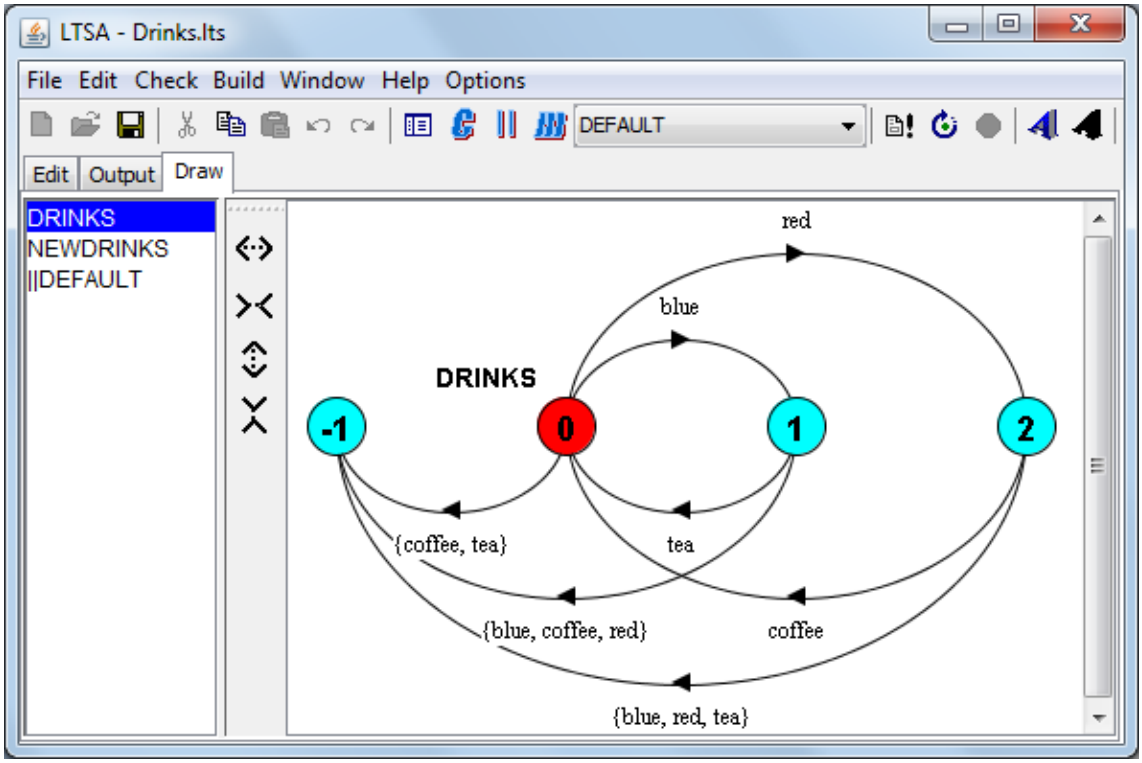


Figura 2.4: Modelo definido como propriedade com o estado de erro.

Ao descrever um novo modelo no mesmo arquivo que a propriedade e uma verificação for realizada, a ferramenta irá comparar o novo modelo com a propriedade definida. O resultado é um novo modelo, onde se espera que todos os comportamentos da propriedade sejam encontrados. Um exemplo de modelo a ser comparado e um exemplo de resultado oriundo da verificação são vistos, respectivamente, nas figuras 2.5 e 2.6.

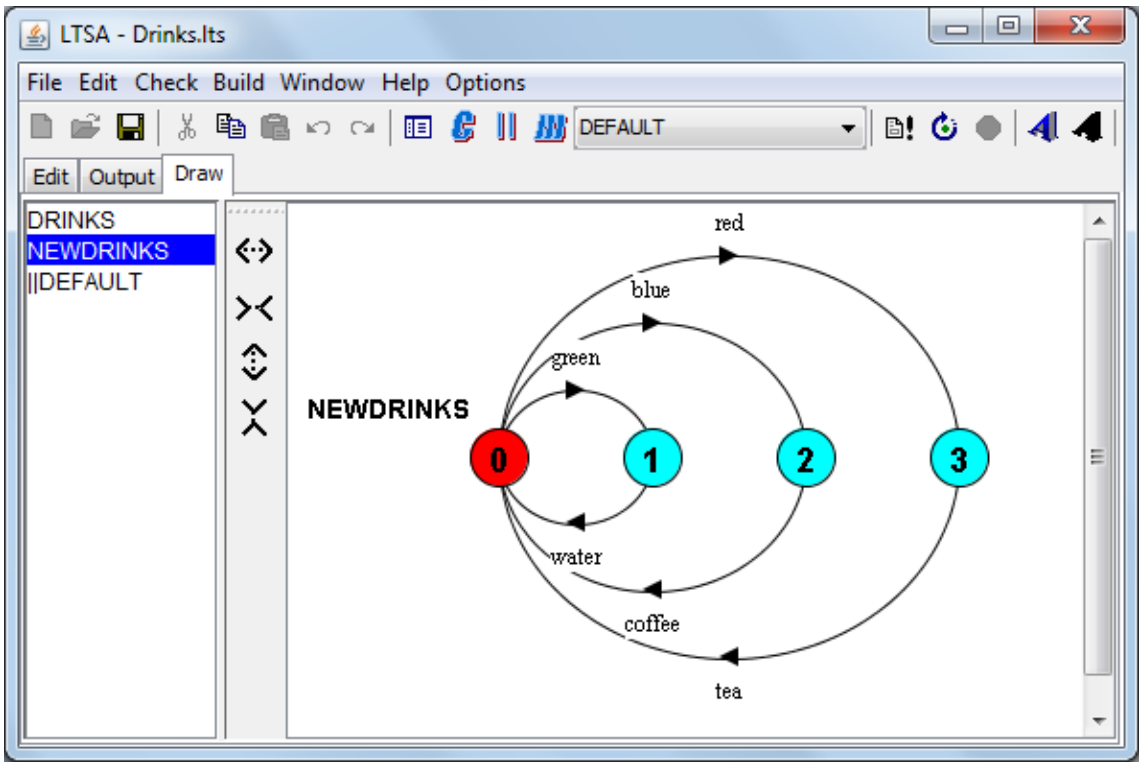


Figura 2.5: Modelo a ser comparado com a propriedade.

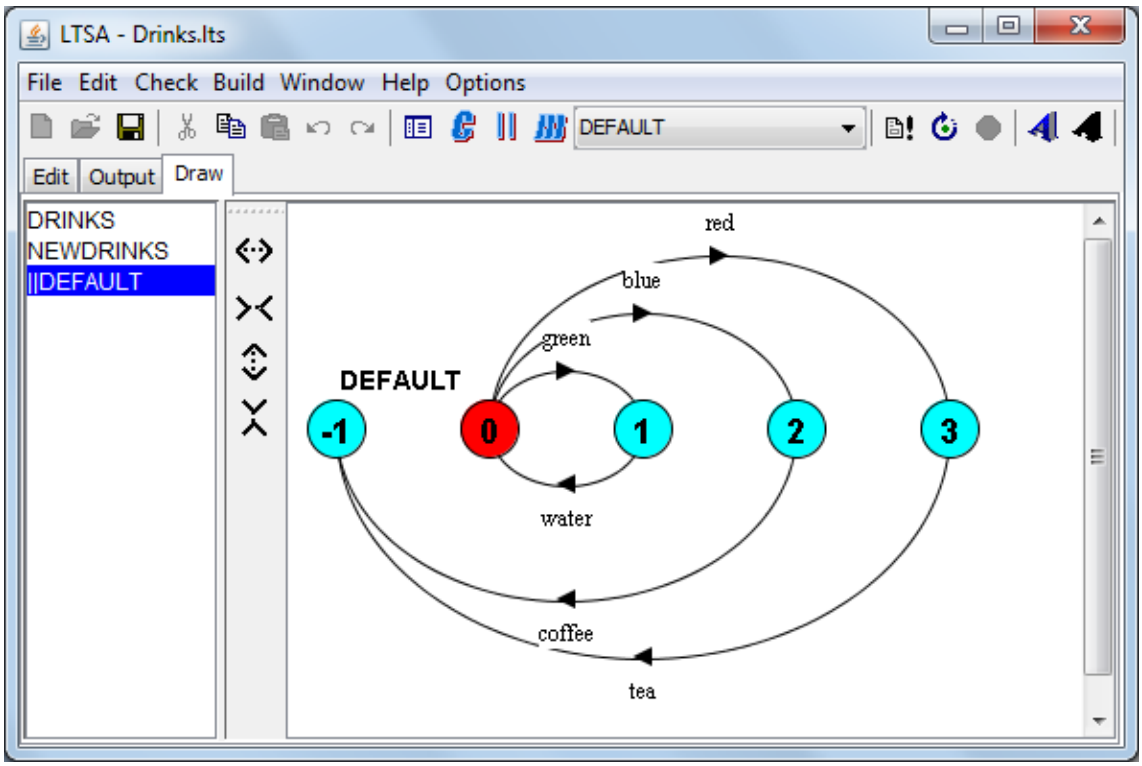


Figura 2.6: Modelo resultante da verificação.

#### *2.2.1.2 LTSE*

O *Labelled Transition System Extractor* (LTSE) é uma ferramenta capaz de extrair informações relevantes à execução de um sistema e descrever um modelo de comportamento baseado em um código fonte Java. Para que o processo funcione, o código deve possuir anotações capazes de descrever possíveis estados do sistema e o fluxo de controle do mesmo. Tais anotações definem um código instrumentado e são criadas utilizando o próprio LTSE, o qual possui uma funcionalidade específica para instrumentação de código. O input desta funcionalidade é o código fonte original do sistema e o output gerado é o código instrumentado.

Para a extração de modelos, a ferramenta utiliza como inputs arquivos de *log* contendo os resultados da execução do conjunto de testes e um arquivo contendo atributos definidos pelo usuário, os quais são relevantes para extrair os comportamentos do sistema. Após o processamento, a ferramenta gera um modelo descrito textualmente, o qual pode ser visualizado no LTSA.

# 3 METODOLOGIA

Neste capítulo, serão descritos os passos realizados em todos os experimentos, após ser definida a alteração na especificação de cada sistema. Para cada sistema, é necessário possuir um modelo de comportamento o qual representa os comportamentos esperados do código e já é considerado correto em relação às propriedades desejadas ao sistema. Além disso, é necessário possuir a implementação de cada sistema, assim como um conjunto de testes previamente criados, os quais possam cobrir os comportamentos relacionados às propriedades.

Após a aplicação da metodologia, teremos como artefatos, um modelo alterado manualmente de acordo com novas propriedades definidas, o código atualizado de acordo com tais modificações propostas e um modelo de comportamento extraído do código, o qual representa os comportamentos encontrados no código com ajuda da extração de modelos e teste de software. Com estes dois modelos, temos como verificar se os comportamentos do modelo manual são encontrados no modelo extraído automaticamente e que comportamentos anteriores às modificações continuam existindo de maneira correta. Desta forma, os objetivos deste trabalho podem ser atingidos.

Neste trabalho, seguiremos a seguinte sequência de passos: alteração manual de modelo, levantamento de casos de teste, instrumentação do código, geração de trace logs, extração de modelo de comportamento e verificação de modelos. Entre a alteração do modelo e o levantamento de casos de teste, é preciso modificar o código, de acordo com a nova especificação. Estes passos poderiam ser realizados em ordens diferentes, podendo produzir resultados diferenciados, baseando-se na hipótese de que alterar a ordem dos passos pode gerar problemas diferentes.

## 3.1 Alteração Manual de Modelo

Dado um modelo estável representando o comportamento extraído do código funcional anterior à modificação, é necessário editar manualmente este modelo através do LTSA, para que os novos comportamentos oriundos da modificação estejam representados neste modelo. Para isso, utiliza-se o LTSA, realizando inclusão, modificação ou exclusão de estados e arcos através do modo editor. Estas modificações são feitas através da descrição textual de estados e arcos, como foi mostrado no capítulo 2.2.

Neste passo, é necessário avaliar o impacto da(s) nova(s) propriedade(s) sobre o modelo existente, pois é necessário que nenhuma propriedade seja desrespeitada após a edição. Após a análise do novo requisito, basta modelar o comportamento esperado e realizar a validação das propriedades. Seria possível definir um conjunto de propriedades para verificar que os comportamentos do modelo são válidos, mas dada a

facilidade permitida pela ferramenta LTSA, é feita uma validação do modelo. Tal validação é uma simulação onde cada estado é visitado ao menos uma vez, analisando quais arcos são executáveis e validando os comportamentos executados contra a especificação definida.

## 3.2 Levantamento de Casos de Teste

Dada à especificação do sistema, criam-se casos de testes, ou modificam-se testes pré-existentes que validem as propriedades estabelecidas, no caso, testes positivos. Para isto, faz-se um levantamento de casos de testes baseado nos requisitos, onde cada teste deve receber uma sequência específica de inputs os quais executem os blocos de código que irão gerar como saída resultados relacionados aos comportamentos descritos pelas propriedades.

Para melhor cobertura de requisitos, é recomendada a criação de testes negativos, capazes de validar como o sistema comporta-se com dados de entrada inválidos. Como a completude da cobertura de testes não faz parte do escopo do trabalho, não há preocupação em se obter um conjunto de testes que seja capaz de cobrir todos os comportamentos possíveis do sistema. É preciso apenas obter um conjunto mínimo de comportamentos que respeitem as propriedades definidas. Por exemplo, não é necessário que testes negativos sejam levantados e executados, caso os testes positivos sejam o suficiente para executar os comportamentos definidos na especificação. Mesmo que não exista a necessidade de uma maior cobertura, não há problema em se obter um conjunto de testes que englobe uma cobertura maior que a mínima necessária.

## 3.3 Instrumentação do Código

Na fase de instrumentação, anotações são adicionadas ao código fonte original, para marcar informações relevantes ao comportamento do sistema, tais quais estruturas de seleção, chamadas de métodos e início e fim de métodos. Além destas informações, o código instrumentado valida valores de certas variáveis relevantes ao fluxo de execução. Com estas informações, é possível identificar diferentes estados que podem compor um modelo em etapas futuras.

A instrumentação é feita de maneira automatizada pelo LTSE, onde uma série de regras é executada sobre o código original, gerando um novo código instrumentado. Um exemplo de código original e instrumentado é mostrado a seguir:

Código original:

```
do {
        try {
                opt = Integer.parseInt(c.readCommand ());
        } catch (IOException e) {
                System.out.println(e.getStackTrace());
        }
} while (opt != END);
  }

  private void greenLights () {
        isGreen = true;
```

```
    changeColour ("green");
  }
```

Código instrumentado:

```
do {
    System.err.println ("REP_ENTER:(opt != END)#" + true + "#" + TrafficLights.class.getName () + "=" + TrafficLights.class.hashCode () + "#" + "{" + "" + "isGreen" + "=" + isGreen + "^" + "}" + "#" + "6" + ";");
    {
     try {
      opt = Integer.parseInt (c.readCommand ());
     } catch (IOException e) {
      System.err.println ("ACTION:" + e.getStackTrace () [0].getClassName ().toLowerCase () + "." + e.getStackTrace () [0].getMethodName () + "_" + e.getClass ().getSimpleName () + "#" + TrafficLights.class.getName () + "=" + TrafficLights.class.hashCode () + ";");
      {
       {
        System.err.println ("CALL_ENTER:getStackTrace" + "#" + TrafficLights.class.getName () + "=" + TrafficLights.class.hashCode () + "#" + e + "#" + "{" + "" + "isGreen" + "=" + isGreen + "^" + "}" + "#" + "1" + ";");
        System.out.println (e.getStackTrace ());
        System.err.println ("CALL_END:getStackTrace" + "#" + TrafficLights.class.getName () + "=" + TrafficLights.class.hashCode () + "#" + e + "#" + "1" + ";");
       }
    System.err.println ("REP_END:(opt != END)" + "#" + TrafficLights.class.getName () + "=" + TrafficLights.class.hashCode () + "#" + "6" + ";");
     }
    }
while (opt != END);
private void greenLights () {
   System.err.println ("MET_ENTER:greenLights" + "#" + TrafficLights.class.getName () + "=" + TrafficLights.class.hashCode () + "#" + "{" + "" + "isGreen" + "=" + isGreen + "^" + "}" + "#" + "7" + ";");
```

```
{
  isGreen = true;
  changeColour ("green");
} System.err.println ("MET_END:greenLights" + "#" + TrafficLights.class.getName () + "=" + TrafficLights.class.hashCode () + "#" + "7" + ";"); }
```

As anotações em negrito visam instrumentar diferentes estruturas do código. O significado de cada anotação é dado abaixo:

- MET_ENTER: Anotação para início da definição de um método.
- MET_END: Anotação para fim da definição de um método.
- CALL_ENTER: Anotação para início da chamada de um método.
- CALL_END: Anotação para fim da chamada de um método.
- REP_ENTER: Anotação para início de uma estrutura de repetição, tal qual while e for.
- REP_END: Anotação para fim de uma estrutura de repetição, tal qual while e for.
- SEL_ENTER: Anotação para início de uma estrutura de seleção, tal qual um if.
- SEL_END: Anotação para fim de uma estrutura de seleção, tal qual um if.
- ACTION: Anotação definidas pelo usuário. Podem ser usadas, por exemplo, após a execução de um dado conjunto de métodos utilizados para a computação de uma tarefa relevante do sistema.

## 3.4 Geração de Trace Logs

O código instrumentado não gera um modelo de comportamento, apenas produz um *trace* da execução atual, isto é, uma lista contendo todos os métodos chamados durante a execução, assim como informações de fluxo de controle. Para adquirir um conjunto relevante de *traces*, a etapa da geração de *logs* é realizada. Para cada caso de teste executado, dois arquivos *log* são gerados, sendo um constituído por um rastro da execução, oriundo das anotações inseridas da etapa de instrumentação, e outro constituído dos outputs do sistema, para melhor validação dos resultados esperados de cada teste.

Abaixo é possível ver um exemplo de *trace log*, com as devidas anotações do código instrumentado apresentado anteriormente.

```
REP_ENTER:(opt!=END)#true#TrafficLightsIntermittent=1868435115#{isGreen=false^lightColor=1^error=false^}#8;

ACTION:in.[1]#TrafficLightsIntermittent=1868435115;

SEL_ENTER:(opt)#RED#TrafficLightsIntermittent=1868435115#{isGreen=false^lightColor=1^error=false^}#7;
```

SEL_ENTER:(lightColor==GREEN)#false#TrafficLightsIntermittent=1868435115#{isGreen=false^lightColor=1^error=false^}#6;

SEL_ENTER:(lightColor==NOCOLOR)#false#TrafficLightsIntermittent=1868435115#{isGreen=false^lightColor=1^error=false^}#4;

SEL_END:(lightColor == NOCOLOR)#TrafficLightsIntermittent=1868435115#4;

SEL_END:(lightColor == GREEN)#TrafficLightsIntermittent=1868435115#6;

SEL_END:(opt)#TrafficLightsIntermittent=1868435115#7;

REP_END:(opt != END)#TrafficLightsIntermittent=1868435115#8;

REP_ENTER:(opt!=END)#true#TrafficLightsIntermittent=1868435115#{isGreen=false^lightColor=1^error=false^}#8;

REP_END:(opt != END)#TrafficLightsIntermittent=1868435115#8;

REP_ENTER:(opt!=END)#false#TrafficLightsIntermittent=1868435115#{isGreen=false^lightColor=1^error=false^}#8;

REP_END:(opt != END)#TrafficLightsIntermittent=1868435115#8;

Pelo exemplo, é possível observar que no código original, há uma estrutura de repetição a qual compara uma variável chamada opt com o valor END. Como a anotação REP_END para esta variável só se encontra no final do exemplo, isto significa que durante a execução do código, o valor de opt era diferente de END. Neste *trace*, vemos uma estrutura de seleção que avalia o valor de opt e logo após avalia o valor de lightColor, o que leva a entender uma sequência de estruturas condicionais aninhadas. O exemplo termina com o término da execução destas estruturas.

## 3.5 Extração de Modelo de Comportamento

O último passo necessário para se obter o modelo necessita que o nome de todos os atributos relevantes do sistema sejam listados. Tais atributos podem assumir diferentes valores, como uma faixa de números inteiros ou um valor booleano, por exemplo. Ao listar todos os atributos em um arquivo, basta utilizá-lo junto aos *trace logs* como argumentos para o LTSE. Ao executar o LTSE, um modelo será formado a partir das informações contidas nos *traces* e arquivo de atributos, o qual será capaz de ser executado pelo LTSA. Através dos *traces* e atributos, o LTSE cria diferentes estados que podem ser encontrados durante o fluxo de execução, os quais farão parte do modelo. Juntando os diferentes *traces*, os estados são interligados, cobrindo os comportamentos do sistema tão bem quanto o conjunto de testes executado pode cobrir. Exemplos de modelos gerados podem ser vistos no capítulo 4, quando os experimentos realizados neste trabalho são apresentados.

## 3.6 Verificação de Modelos

Utilizando o LTSA, é possível visualizar um ou mais modelos, assim como executar uma simulação de passeios por seus caminhos. Através desta simulação, é possível identificar para cada estado do modelo, quais são os possíveis arcos que partem do mesmo. Com dois modelos, é possível realizar uma comparação para verificar se os comportamentos descritos em um dos modelos são mantidos em outro. Assim, podemos

observar os comportamentos de um modelo extraído do código estão de acordo com um modelo alterado manualmente.

Para comparar comportamentos entre modelos, toma-se um modelo como base, e com o comando *property*, dizemos ao LTSA que aquele modelo escolhido será utilizado para comparação. Isto é feito adicionando-se a palavra especial *property* na primeira linha da descrição textual do LTS visualizado (como foi mostrado na figura 2.3). O LTSA cria um estado de erro, com rótulo -1. Além disso, são adicionados arcos em todos os estados válidos do modelo, com destino ao estado de erro, para representar todos os comportamentos que o modelo não possui originalmente, dados os valores de entrada e métodos descritos.

Ao executar um modelo no LTSA utilizando outro como propriedade, um novo modelo é gerado. Caso todos os comportamentos da propriedade estejam corretamente incluídos no modelo comparado, o resultado gerado é um modelo que não possui um estado de erro. Caso o estado de erro seja encontrado no resultado, isto significa que o modelo comparado possui comportamentos os quais não estão de acordo com a propriedade. Os comportamentos considerados incorretos, quando confrontados com a propriedade, são identificados no modelo através dos arcos cujo destino é o estado de erro.

A verificação de forma automática e com representação gráfica permite que sejam identificados facilmente discrepâncias de comportamentos entre dois modelos, pois a presença do estado de erro no modelo resultante da verificação indica que há diferenças no comportamento de ambos os modelos verificados. Assim, é possível validar através de modelos se uma ou mais alterações feitas inseriram erros nos comportamentos já existentes do sistema.

# 4 EXPERIMENTOS

Para estudar as vantagens e desvantagens de modelos gerados manualmente ou automaticamente, serão feitas modificações em sistemas que já possuem especificação e código. Um conjunto inicial de testes é levantado para cada sistema e, de maneira incremental, são adicionadas novas propriedades ou funcionalidades às especificações. Para cada mudança, o modelo existente é alterado manualmente, de forma a refletir todos os comportamentos esperados. O código é alterado para cobrir os mesmos comportamentos, e então todas as etapas descritas na seção 2.2.2 são utilizadas.

## 4.1 Traffic Lights

O exemplo de semáforo utilizado para iniciar os experimentos consiste em um tipo simples de semáforo, o qual possui apenas dois estados visualizados pelo usuário: luz de cor vermelha e luz de cor verde. Esta implementação possui apenas uma propriedade a ser seguida, a qual consiste na alternância entre as cores vermelho (Red) e verde (Green). As saídas visualizadas pelo usuário final consistem apenas nas duas cores, as quais são acionadas por comandos numéricos fornecidos na chamada do programa.

Especificação:

Propriedade 1: As luzes do semáforo devem intercalar entre as cores vermelho e verde.

property CorrectLights = (greenLights -> redLights -> CorrectLights).

Código:

```
import java.io.IOException;

/**
  * Implements the traffic lights system behaviour.
  *
  * @author Lucio Mauro Duarte
  * @version 26/10/2011
  */
class TrafficLights {
```

```
  private static final int GREEN = 0;
  private static final int RED = 1;
  private static final int END = 2;
  private static boolean isGreen;

 public TrafficLights (CommandReader c) {
  isGreen = false;
  int opt = - 1;
  do {
     try {
        opt = Integer.parseInt(c.readCommand ());
     } catch (IOException e) {
        System.out.println(e.getStackTrace());
     }
     //#input:"["+opt+"]";
     switch (opt) {
      case GREEN :
       if (!isGreen)
          greenLights ();
      break;

     case RED :
       if (isGreen)
          redLights ();
      break;
     }
  } while (opt != END);
 }

 private void greenLights () {
    isGreen = true;
    changeColour ("green");
 }

 private void redLights () {
  isGreen = false;
  changeColour ("red");
 }

 private void changeColour (String newColour) {
    System.out.println (newColour);
 }
}
```

A partir da especificação do sistema, foi criado um conjunto inicial de testes para garantir que o requisito estipulado é respeitado. Os testes gerados são do tipo caixa preta, sem influência do código e considerando apenas a especificação, entradas e saídas.

Para a criação do conjunto inicial de testes, foram consideradas as possíveis transições entre a cor verde e vermelha, incluindo nenhuma transição entre estados. Dado o pequeno número de possibilidades, foi possível listar todas as transições que o sistema deve permitir ou barrar.

Considerando-se testes positivos e negativos, o conjunto de casos de teste criado foi:

**T1: GreenToGreen**:

- Descrição: Testa o comportamento do sistema quando a cor verde está ativada e o comando para acionar a cor verde é executado.
- Vetor de inputs: [0, 0, 2]
- Resultado Esperado: Cor verde não tem transição para a cor verde. O usuário não deve ver mudanças nos *logs* de saída do sistema.

**T2: GreenToRed**:

- Descrição: Testa o comportamento do sistema quando a cor verde está ativada e o comando para acionar a cor vermelha é executado.
- Vetor de inputs: [0, 1, 2]
- Resultado Esperado: Cor vermelha alterna após a cor verde. O usuário deve ser capaz de ver o resultado [green, red] nos *logs* de saída do sistema.

**T3: RedToGreen**:

- Descrição: Testa o comportamento do sistema quando a cor vermelha está ativada e o comando para acionar a cor verde é executado.
- Vetor de inputs: [0, 2]
- Resultado Esperado: Cor verde alterna após a cor vermelha. O usuário deve ser capaz de ver o resultado [red, green] nos *logs* de saída do sistema.

**T4: RedToRed**:

- Descrição: Testa o comportamento do sistema quando a cor vermelha está ativada e o comando para acionar a cor vermelha é executado.
- Vetor de inputs: [1, 2]
- Resultado Esperado: Cor vermelha não tem transição para a cor vermelha. O usuário não deve ver mudanças nos *logs* de saída do sistema.

**T5: Exit**:

- Descrição: Testa o comportamento do sistema quando o comando para finalizar o programa é executado.
- Vetor de inputs: [2]

- Resultado Esperado: Fim de execução. Nada é mostrado nos *logs* de saída do sistema.

Para a construção do modelo a partir do código, o conjunto inicial de testes foi executado uma vez, gerando *logs* com informações sobre o comportamento do sistema. Estes *logs* foram utilizados com a ferramenta LTSE (Labelled Transition System Extractor), com o objetivo de construir a representação gráfica do LTS.

Modelo 1: Traffic Lights com as cores Red e Green, extraído do código do sistema.

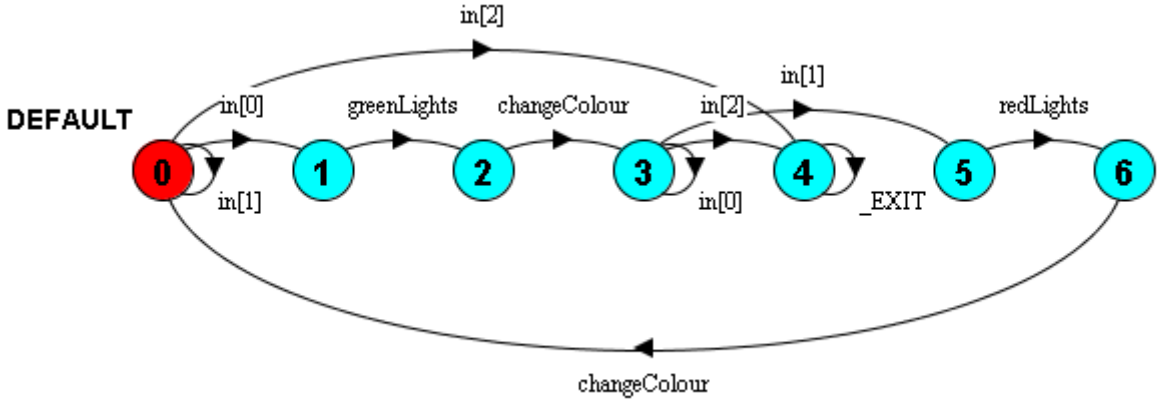


Figura 4.1: Modelo 1 - Traffic Lights com as cores Red e Green, extraído do código do sistema.

Pelo modelo, é possível ver os possíveis estados:

| Estado | Descrição |
|---|---|
| 0 | Semáforo está ligado na cor vermelha. Os possíveis comandos são: |
| | 0: Cor verde. Sistema passa para o estado 1. |
| | 1: Cor vermelha. O sistema permanece no mesmo estado e não gera saída alguma. |
| | 2: Final de programa. A execução termina e o sistema passa para o estado 4. |
| 1 | Comando 0 foi lido. O sistema executa o método greenLights, passando para o estado 2. |
| 2 | Método greenLights é executado. O sistema passa automaticamente para o estado 3. |
| 3 | Método changeColour é executado e a cor do semáforo passa a ser verde. Os possíveis comandos são: |
| | 0: Cor verde. O sistema permanece no mesmo estado e não gera saída alguma. |
| | 1: Cor vermelha. Sistema passa para o estado 5. |
| | 2: Final de programa. A execução termina e o sistema passa para o estado 4. |
| 4 | Fim de execução. |
| 5 | Comando 1 foi lido. O sistema passa automaticamente para o estado 6. |
| 6 | Método redLights é executado. O sistema passa automaticamente para o estado 0. |

Tabela 4.1 – Descrição dos estados e arcos do modelo 1

De acordo com a validação realizada no LTSA, o modelo 1 satisfaz a propriedade de alternância de cores, não permitindo que o mesmo valor de input ao ser chamado duas ou mais vezes seguidas execute mais de uma vez o método respectivo (redLights e greenLights). Para este modelo, somente é possível executar a chamada de green após red e de red após green, salvo o estado inicial, que já começa com a cor red acesa.

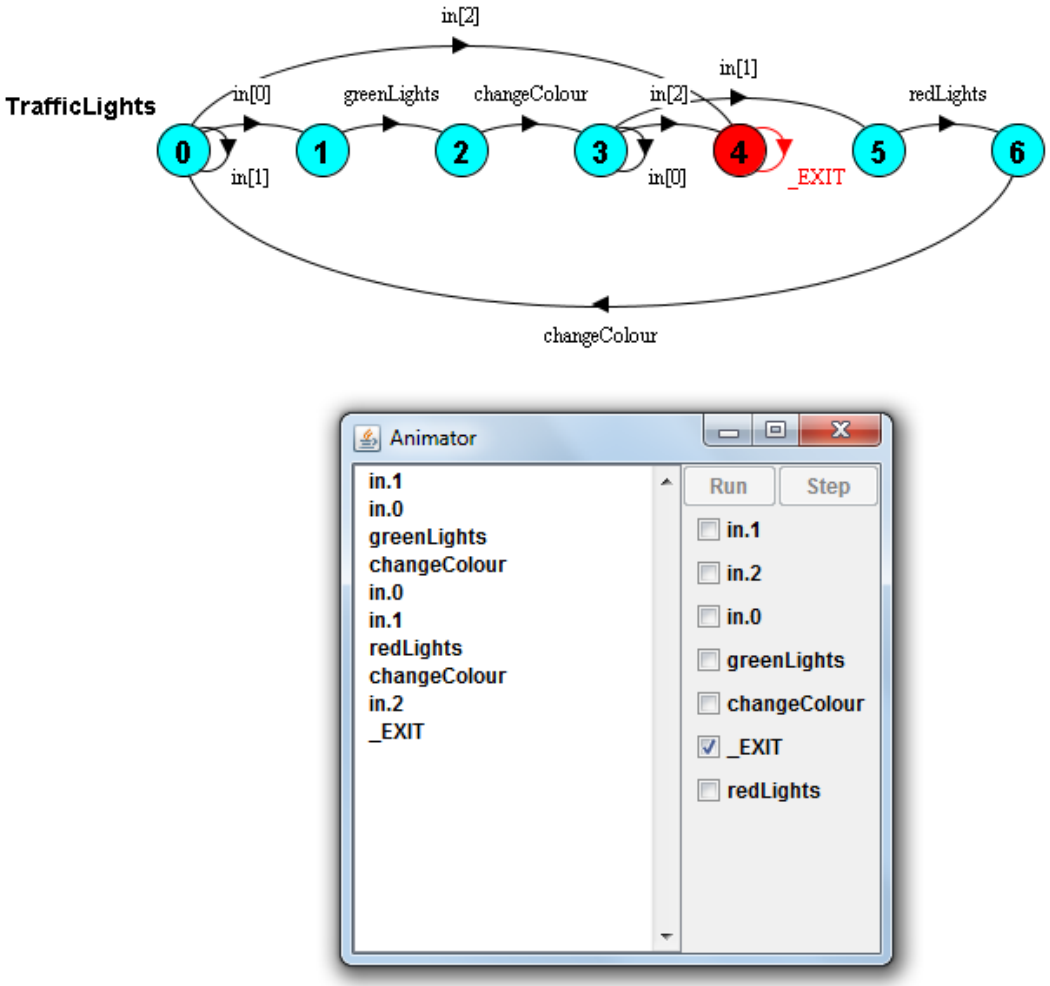


Figura 4.2: Validação das transições entre estados permitida pela ferramenta LTSA.

Como primeiro experimento, estudou-se a modificação da propriedade, a qual obriga uma mudança no código e modelo. Foi adicionada a cor amarela e o requisito do sistema passou a obrigar a transição da cor verde para amarela e amarela para vermelha. A cor vermelha continua tendo transição direta para a cor verde. Este comportamento reflete os estados e transições comuns aos semáforos reais encontrados nas cidades.

Propriedade 1:

property CorrectLights = (greenLights -> yellowLights -> redLights -> CorrectLights).

Neste experimento dois novos modelos foram gerados. Um manualmente, de forma a respeitar as propriedades estabelecidas independentemente do código, e um modelo gerado pela ferramenta LTSE, após o código original ser alterado para refletir o comportamento do semáforo de acordo com a nova especificação. Seguindo a estrutura do código, um novo método yellowLights foi criado, semelhante ao greenLights e redLights já existentes. Esse método yellowLights somente é executado quando há uma transição da cor verde para a cor vermelha, não sendo necessário que o usuário explicitamente entre com um valor de input específico para executar a cor amarela.

O modelo 1 foi modificado manualmente para que as mudanças propostas fossem refletidas sem alteração no código, para que o comportamento esperado possa ser validado pelo modelo.

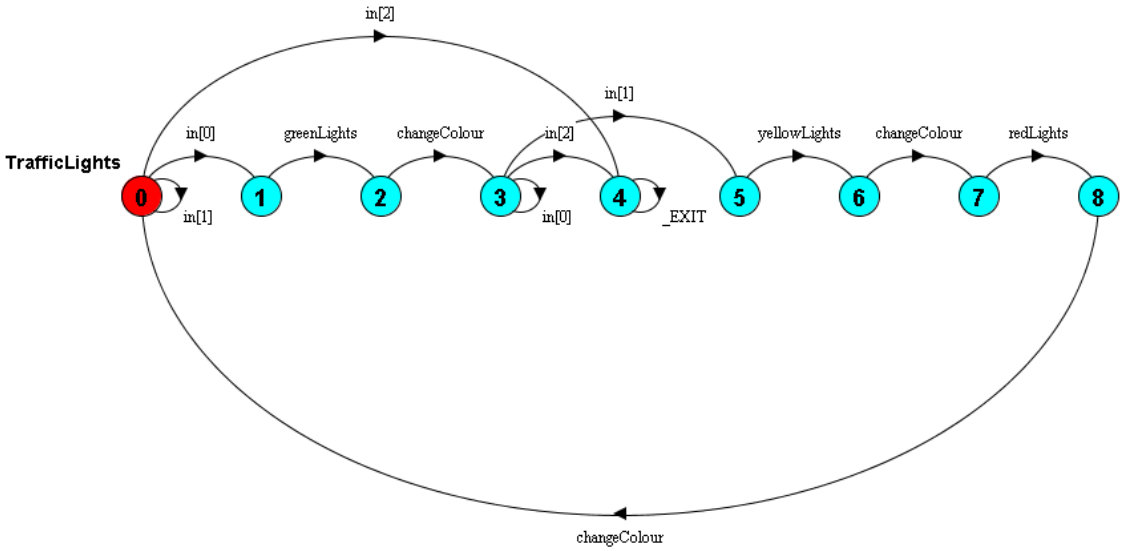


Figura 4.3: Modelo 2 - Traffic Lights com as cores Red, Yellow e Green, alterado manualmente.

Neste modelo, dois novos estados e transições foram adicionados. Após a chamada de changeColour gerada por greenLights, foi adicionada a transição yellowLights, seguida da transição changeColour (transições entre os estados 5, 6 e 7). Com essa

mudança, o semáforo obrigatoriamente deve mostrar a cor amarela quando o sistema está no estado onde a luz verde está ligada e recebe uma entrada para passar ao estado com a luz vermelha.

Para modificar o código original, de acordo com a nova especificação, criou-se um método semelhante a greenLights e redLights, indicando que o a cor na qual o semáforo deve estar ligado no momento é a cor amarela. A chamada deste método é feita somente quando o semáforo está mostrando a luz verde e recebe como input o comando para exibir a cor vermelha.

Código alterado:

```
private static int lightColor;

lightColor = RED;

switch (opt) {
        case GREEN :
         {

          {
           {
            if (lightColor == RED) {
              greenLights ();
            }

            else if(lightColor == GREEN) {
              }

            }

          } break;

         }

        case RED :
         {
          {
           {

            if (lightColor == GREEN) {
              yellowLights ();
              redLights ();
            }

            else if(lightColor == RED) {
              }

             }
```

```
            } break;

          }
      }

  private void greenLights () {
    {
     lightColor = GREEN;
     isGreen = true;
     changeColour ("green");
    }
  }

  private void redLights () {
    {
     lightColor = RED;
     isGreen = false;
     changeColour ("red");
    }
  }

  private void yellowLights () {
    {
     lightColor = YELLOW;
     changeColour ("yellow");
    }
  }
```

Um novo modelo foi gerado utilizando-se a mesma metodologia e ferramentas. O conjunto de testes permaneceu inalterado pois as entradas não influenciam diretamente no estado onde o semáforo deve permanecer com a cor amarela acesa. O teste da transição de verde para vermelho (T2: GreenToRed) deve mostrar a cor amarela nos novos resultados, utilizando as mesmas entradas do experimento anterior.

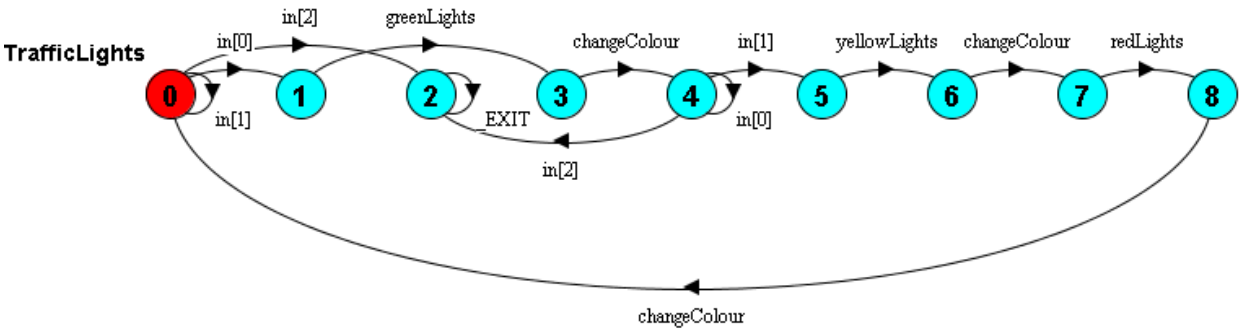


Figura 4.4: Modelo 3 - Traffic Lights com as cores Red, Yellow e Green, extraído do código alterado do sistema.

O modelo 3, assim como o modelo 2, caracteriza-se pela inclusão do método yellowLights (arco ligando estado 3 ao estado 4 no modelo 2 e estado 5 ao estado 6 no modelo 3), executado após o input 1 ser processado pelo sistema quando o semáforo encontra-se ligado na cor verde (estado 4), seguido de uma chamada do método changeColour.

Aceitando que, após a validação, o modelo gerado manualmente é correto de acordo com a especificação, ele será usado como uma propriedade na ferramenta LTSA e comparado com outros modelos.

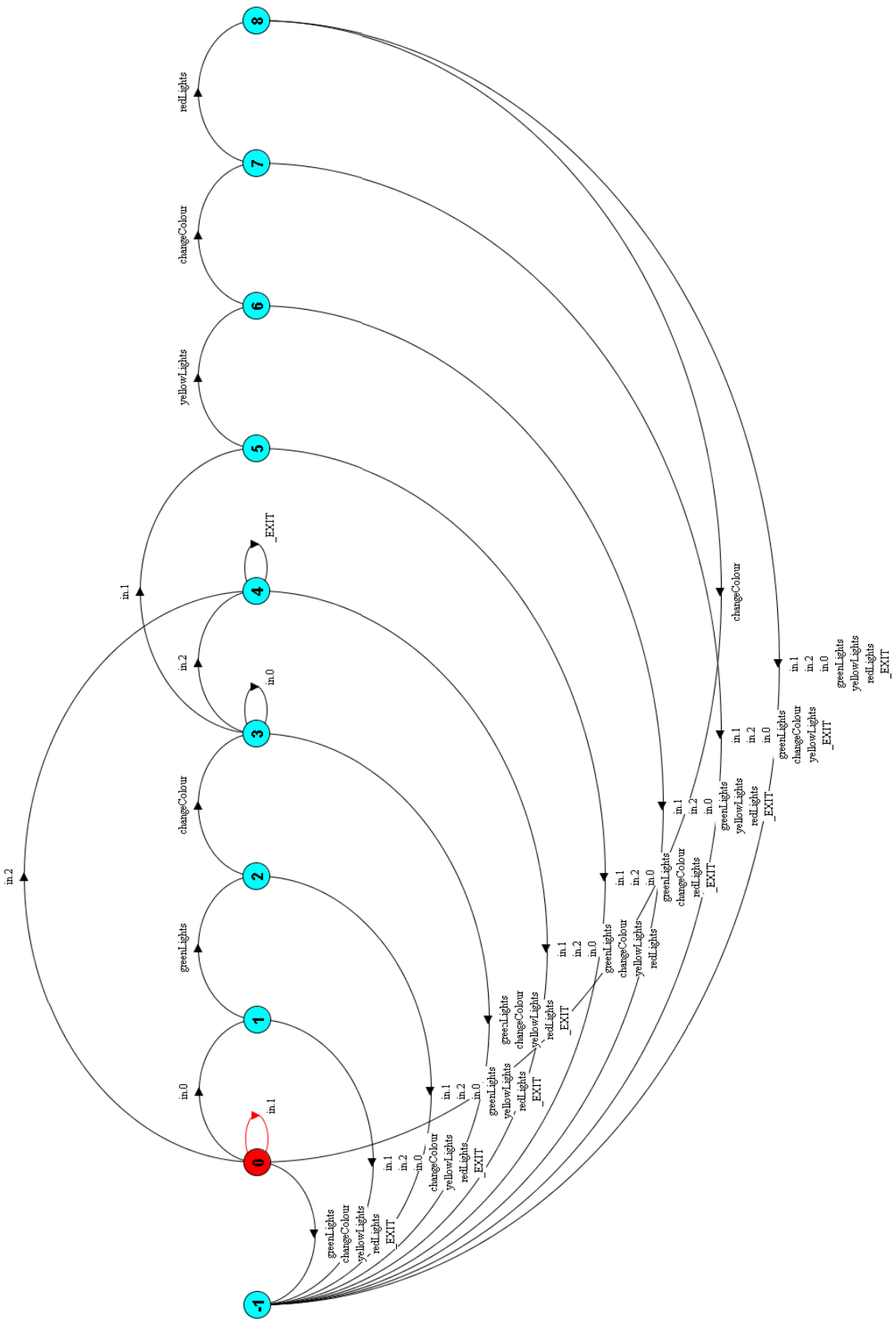


Figura 4.5: Modelo definido como propriedade

Após a verificação dos modelos no LTSA, concluiu-se de que transição de cores definida pela especificação do sistema é respeitada, pois nenhum arco resultante tem como destino um estado de erro.

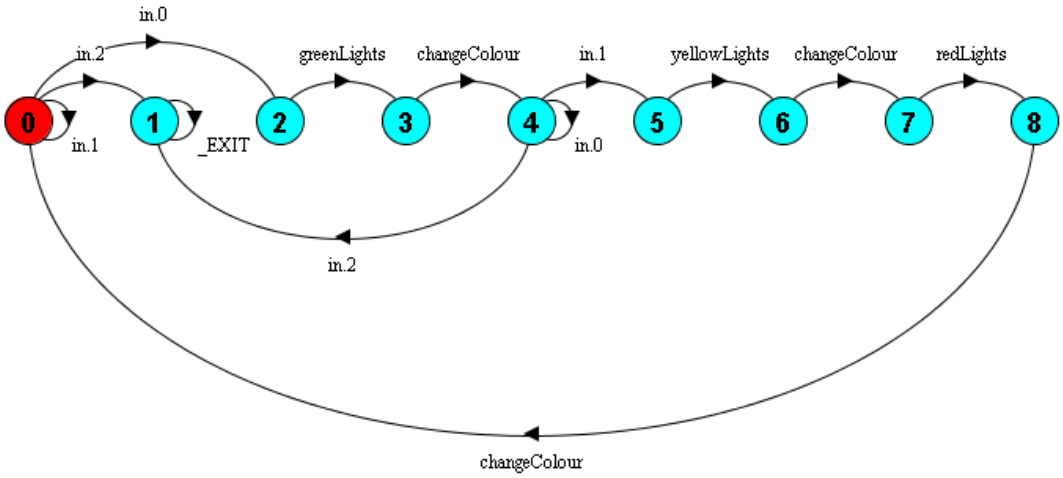


Figura 4.6: Verificação dos modelos 2 e 3

A última alteração adicionada permite que o semáforo entre em um estado de erro, caso não receba como entrada as opções verde (0), vermelho (1), ou fim de execução (2). Este estado de erro é representado por uma luz amarela intermitente. Além disso, o semáforo poderá ser reiniciado quando se encontrar em um estado de erro, voltando assim a um estado válido para execução.

Propriedade 2:

property ErrorState = (yellowLights-> turnOffLights -> ErrorState).

Propriedade 3:

property Restart = (turnOffLights-> redLights -> Restart).

Para o modelo gerado manualmente, foi pensado em dois possíveis agrupamentos, onde o semáforo encontra-se em estados válidos ou em um estado inválido. A partir de modelos anteriores, foi adicionado o agrupamento representando as novas propriedades, onde a luz amarela intermitente é acionada e o semáforo pode ser reiniciado para seu estado inicial original. Como a implementação não foi levada em consideração, os arcos que levam ao estado de erro é chamado genericamente de “error”. Aproveitando a lógica de modelos anteriores, quando a luz deve ser desligada para simular a intermitência, usa-se o arco changeColour, supondo-se que neste momento, a cor será mudada para “nenhuma cor”, ou seja, o semáforo estará com as luzes desligadas.

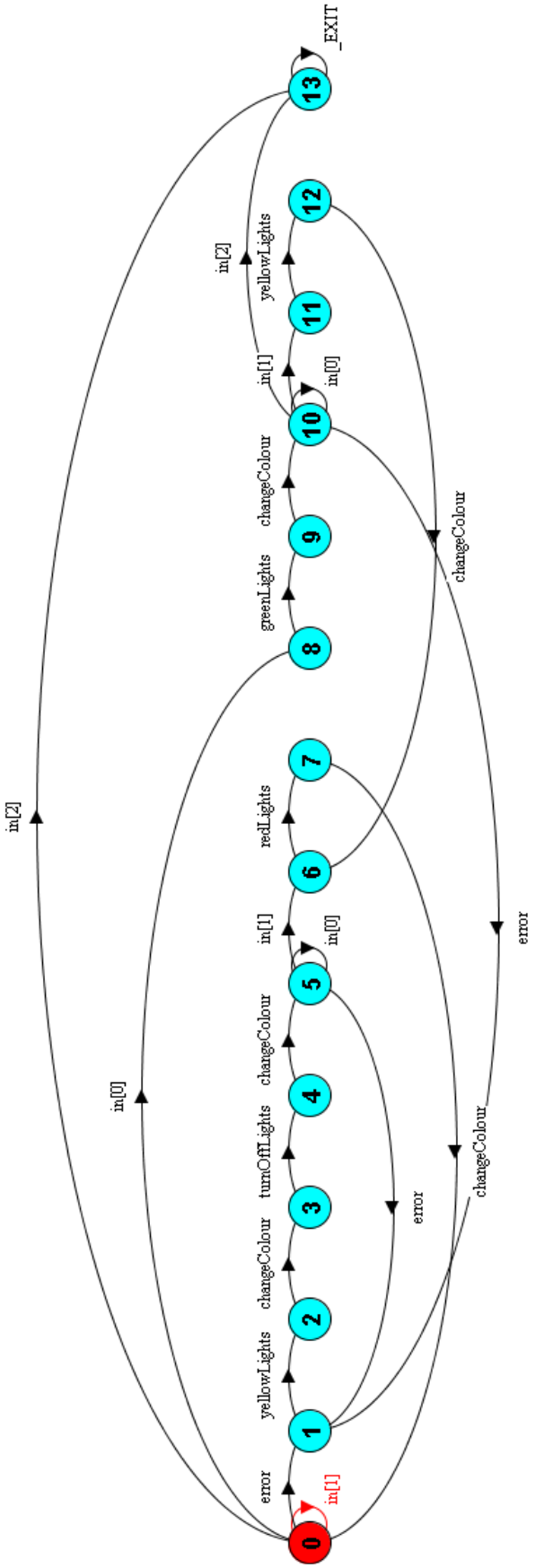


Figura 4.7 : Modelo 4 – Luz intermitente - Modelo gerado manualmente

Para a implementação do sistema, sem considerar os modelos anteriores. Utilizou-se a estrutura switch que controla as entradas do sistema, para ativar o estado inválido do semáforo para cada valor numérico não esperado como valor válido de entrada. Quando o semáforo encontra-se no estado de intermitência, uma variável booleana error tem seu valor alterado para true, voltando ao valor original, false, quando o semáforo é reiniciado.

Código:

```
switch (opt) {
        case GREEN :
          if (lightColor == RED)
          {
             yellowLights ();
             greenLights ();
          }else if (lightColor == NOCOLOR)
          {
             error = false;
             greenLights ();
          }
         break;

        case RED :
          if (lightColor == GREEN)
          {
             yellowLights ();
             redLights ();
          }else if (lightColor == NOCOLOR)
          {
             error = false;
             redLights ();
          }
         break;

         default:
            error = true;
            yellowLights();
            turnOffLight();
              break;
       }

private void turnOffLight () {
     lightColor = NOCOLOR;
     changeColour ("No Color");
}
```

Como o processo de criação do modelo necessita de um conjunto de testes, foi necessário criar novos casos de teste para incorporar as novas propriedades. Assim como o conjunto inicial, os novos testes são criados de forma a validar propriedades definidas. Para cobrir os comportamentos esperados pela nova propriedade, dois testes foram levantados para validar o comportamento do sistema ao entrar e ao sair do estado onde a luz amarela deve piscar intermitentemente. Assim temos:

**T6: ErrorState**:

- Descrição: Testa o comportamento do sistema quando um comando inválido é executado.
- Vetor de inputs: [5, 6, 7, 8, 9, 2]
- Resultado Esperado: O semáforo deve piscar a cor amarela intermitentemente. O usuário deve ver a saída [Yellow, No Color] nos *logs* de saída do sistema.

**T7: RestartTrafficLight**:

- Descrição: Testa o comportamento do sistema quando comandos inválidos são executados anteriormente a um reinício do sistema.
- Vetor de inputs: [3, 4, 5, 6, 7, 8, 9, 1, 3, 4, 5, 6, 7, 8, 9, 0, 3, 4, 5, 6, 7, 8, 9, 0, 3, 4, 5, 6, 7, 8, 9, 1, 2]
- Resultado Esperado: O sistema deve piscar a luz amarela intermitentemente e ser reiniciado quando o comando de luz vermelha é executado.

A escolha da implementação, onde o estado de erro é gerenciado na estrutura *default* do comando *switch*, somada aos testes escritos de forma incompleta, gerou um modelo que satisfaz parcialmente as condições estabelecidas, pois não é possível obter no modelo, todos os possíveis valores de entrada inválidos para que o semáforo pisque a luz amarela de forma intermitente. Como o processo de automatização diferencia a transição dos estados verde e vermelho para o estado inválido, o modelo gerado automaticamente possui duas áreas onde a luz amarela intermitente é ativada.

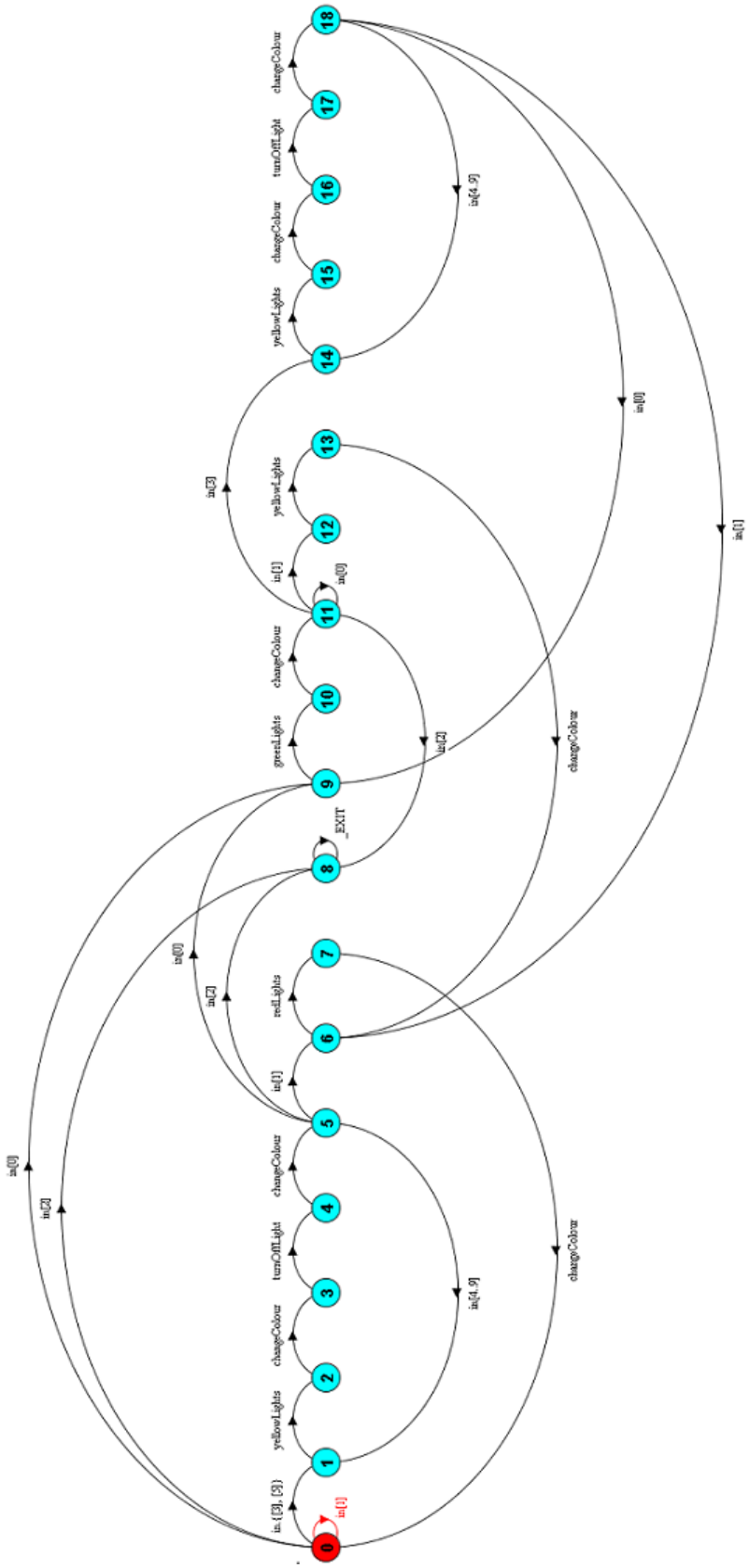


Figura 4.8: Modelo 5 – Luz Intermitente Modelo gerado automaticamente, implementação sem exceção

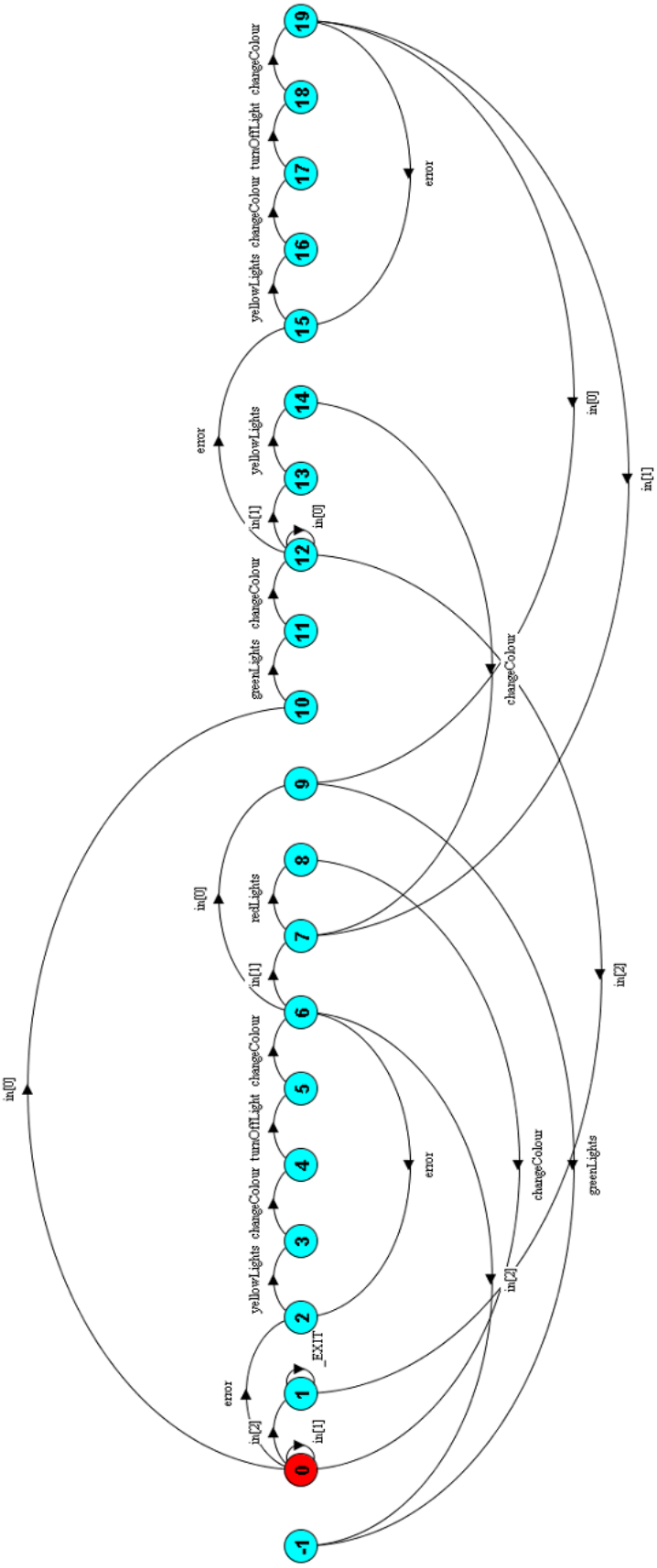


Figura 4.9: Verificação dos modelos 4 e 5

Ao término da verificação dos modelos, observou-se que a implementação não seguiu todos os comportamentos esperados. Dois arcos com destino ao estado de erro foram gerados, de tal forma que é possível terminar a execução do sistema quando o mesmo encontra-se em intermitência, e a possibilidade de reiniciar o semáforo com a luz verde, caso o mesmo encontre-se em um estado de intermitência.

Baseado no modelo manual, utilizou-se uma exceção para executar os métodos que farão o semáforo entrar no estado de intermitência, já que esta estrutura sugere uma "área separada do código" para gerenciar os inputs os quais devem ativar a intermitência da cor amarela. Por esta implementação, o modelo conterá arcos exception, ao invés de arcos com as possíveis entradas inválidas, semelhante à lógica dos arcos error, encontrados no modelo manual. Este conceito é genérico o suficiente para abranger valores não esperados e mantém um modelo mais simples e legível, que possui apenas uma área com estados inválidos. É notável a semelhança com o modelo manual no qual foi inspirado, o que gerou um processo mais fácil e rápido entre codificação e validação de resultados corretos.

Código com exceção:

```
try {
          opt = Integer.parseInt (c.readCommand ());
        } catch (IOException e) {
        } catch (NumberFormatException e) {
         {
          error = true;
          yellowLights ();
          turnOffLight ();
         }


       switch (opt) {
        case GREEN :
             {
              if (lightColor == RED) {

                 greenLights ();
               }


             }


          break;

        case RED :
```

```
            {
             if (lightColor == GREEN) {

                yellowLights ();
                redLights ();
              }

             else
              {
               if (lightColor == NOCOLOR) {

                  error = false;
                  redLights ();
                }
              }
          }
          break;
    }


  private void turnOffLight () {
     lightColor = NOCOLOR;
     isGreen = false;
     changeColour ("No Color");
  }
```

Os mesmos testes T6 e T7 foram utilizados para este processo, com a diferença dos vetores de entrada, que passaram a ser [ , ,2] e [ ,1, ,0,2]. O espaço em branco foi utilizado para que a exceção utilizada fosse chamada e este comportamento fosse adicionado ao modelo.

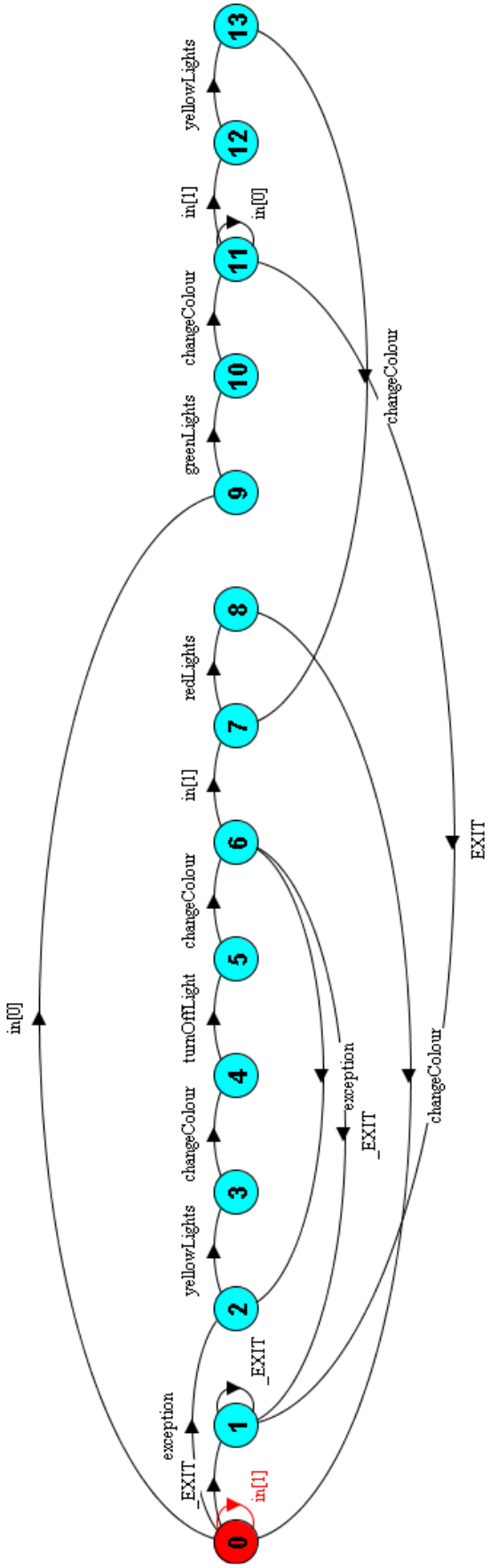


Figura 4.10: Modelo 6 - Luz Intermitente - Modelo gerado automaticamente, implementação utilizando exceção

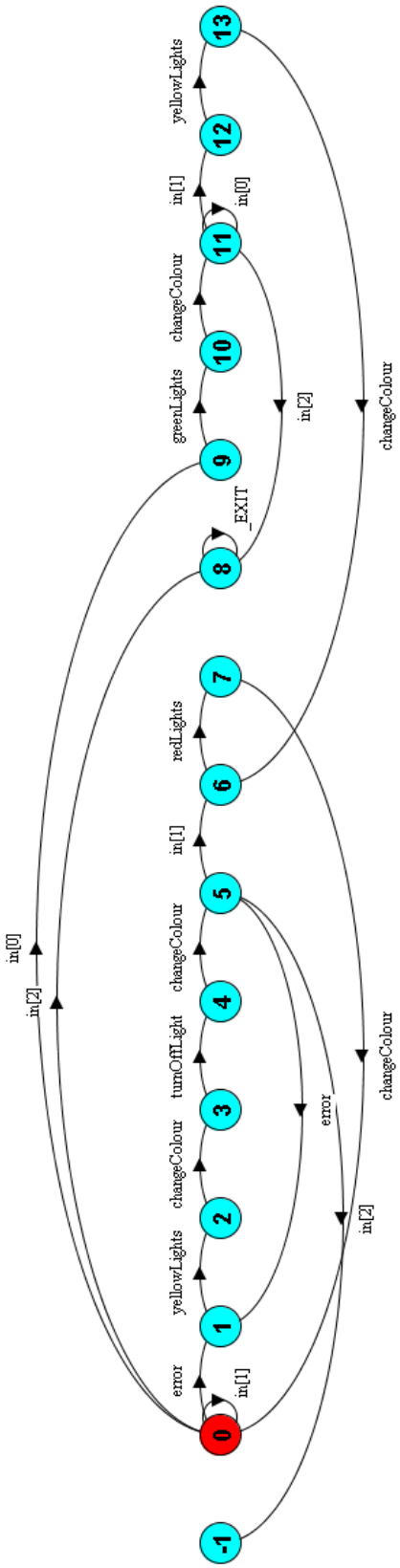


Figura 4.11 Verificação dos modelos 4 e 6

Nesta validação, verificou-se apenas a possibilidade de terminar a execução do código durante um estado de intermitência. Segundo as validações, os modelos automatizados não tem o comportamento esperado pelo modelo gerado manualmente, mas este último não se encontra totalmente correto. A falha humana foi não prever o término do sistema durante um estado de intermitência, já que o fim de execução é uma ação válida no momento em que o código lê um novo valor de entrada. Corrigindo este problema, temos um modelo gerado manualmente e um modelo gerado a partir do código fonte representando os mesmos comportamentos descritos na especificação.

| Traffic Lights | | | |
|---|---|---|---|
| | Número de Estados | Número de Arcos | Número de Erros |
| Modelo 2 | 9 | 13 | 0 |
| Modelo 3 | 9 | 13 | 0 |
| Modelo 4 | 14 | 22 | 1 |
| Modelo 5 | 19 | 30 | 1 |
| Modelo 6 | 14 | 21 | 0 |

Tabela 4.2: Resultados dos experimentos realizados com Traffic Lights

Ao analisar cada experimento separado, chegou-se às seguintes conclusões:

- Acréscimo da cor amarela: Tendo em vista que a modificação proposta era muito simples, não houve maior dificuldade entre alterar o modelo ou o código. Ambas as formas foram executadas em pouco tempo e sem muita complexidade. Os modelos resultantes foram satisfatórios e muito semelhantes, possuindo o mesmo número de estados e transições.

- Acréscimo da cor amarela intermitente: A modificação manual do modelo foi mais rápida e simples de ser executada. Por se tratar de uma funcionalidade separada do comportamento normal do sistema, foi possível modelar a intermitência da cor amarela em uma área à parte do modelo, ligando esta área através de arcos aos estados que podem dar origem ao estado de intermitência. As implementações feitas foram simples, porém não tão óbvias quanto à alteração manual. Utilizar o default do switch para ativar a intermitência é uma alteração trivial, mas não é possível observar todas as entradas que levam a este estado no modelo, já que se tratam de infinitas possibilidades. A utilização de exceções resultou em um modelo melhor, mais parecido com o alterado manualmente, porém foi necessário tomar cuidado ao codificar a reinicialização do semáforo a partir do estado de intermitência.

O resultado obtido no segundo experimento mostra como a maneira a qual uma modificação é implementada influi no modelo final. Utilizando o modelo manual como guia para modificar o código, é possível gerar um modelo menos redundante e mais legível.

## 4.2 Editor

A versão inicial do editor de texto usado consiste em cinco funções básicas agindo sobre apenas um documento por execução: open, print, edit, save e close. Algumas propriedades são descritas para garantir que certas operações possam ocorrer apenas após outras operações específicas, e junto com as próprias operações, formam a especificação deste sistema.

As saídas visualizadas pelo usuário final consistem das operações utilizadas, as quais são acionadas por comandos numéricos fornecidos na chamada do programa.

Especificação:

Propriedade 1: Todo documento aberto deve ser fechado

```
property OpenAndClose = CLOSED,
CLOSED = (open -> OPEN),
OPEN = (close -> CLOSED).
```

Propriedade 2: Um documento pode ser salvo apenas após ser editado

```
property SaveOnlyIfEdited = SAVED,
SAVED = (edit -> EDITED),
EDITED = (edit -> EDITED
      |save -> SAVED).
```

Propriedade 3: Um documento pode ser editado, impresso ou salvo apenas após ter sido aberto.

```
fluent Open = <open, close> initially 0
assert Allowed = (!Open -> !(edit || print || save))
```

Código:

```
/**
  * Implements the editor system.
  *
  * @author Lucio Mauro Duarte
   * @version 09/02/2011
  */
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);

     //#input:"["+cmd+"]";
     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;

          default :
             System.out.println ("\nIncorrect command");
            //#action:"incorrectCmd";
       }
       } catch (Exception e) {
        System.out.println ("\nIncorrect command");
        //#action:"incorrectCmd";
     }
   }
}

void open () {
    isOpen = true;
  System.out.println ("\n> File opened");
}

void edit () {
  isSaved = false;
  System.out.println ("\n> File edited");
}

void print () {
  System.out.println ("\n> File printed");
}

void save () {
  isSaved = true;
  System.out.println ("\n> File saved");
}

void close () {
  isOpen = false;
  System.out.println ("\n> File closed");
}

void exit () {
      if (!isSaved) {
    System.out.print ("\n> Save modifications? ");
    int opt = 0;
    if (opt == 0)
        save ();
       }

  if (isOpen)
     close ();

  System.out.println ("\n> Editor exiting...");
```

```
  }
}
```

Considerando-se testes positivos e negativos, o conjunto inicial de testes criado para o Editor foi:

**T1: Operação open-close**:

- Descrição: Permite que um documento seja fechado imediatamente após ser aberto.
- Vetor de inputs: [0, 4]
- Resultado Esperado: O documento é aberto e fechado em sequência.

**T2: Operação print:**

- Descrição: Permite que a operação print possa ser executada em qualquer momento entre as operações open e close.
- Vetor de inputs: [0, 2, 1, 2, 3, 2, 4]
- Resultado Esperado: Operação print é utilizada em todas as chamadas do método print().

**T3: Operações em sequência**:

- Descrição: Testa o uso de todas as operações em sequência: open, edit, print, save, close.
- Vetor de inputs: [0, 1, 2, 3, 4]
- Resultado Esperado: Todas as operações são corretamente executadas na sequência chamada.

Após uma rodada de testes, foram gerados *logs* para extrair o comportamento do sistema, estes utilizados na geração do modelo final.

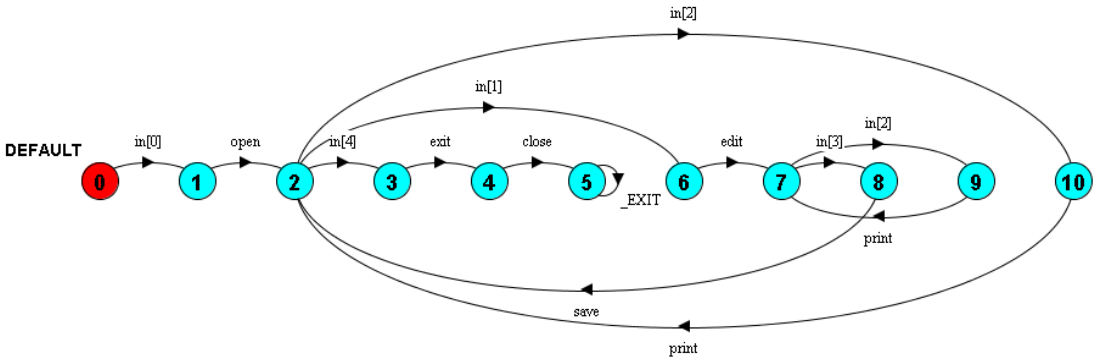


Figura 4.12: Modelo 1: Editor com propriedades e operações originais, extraído do código do sistema.

Pelo modelo, é possível ver os possíveis estados:

| Estados | Descrição |
|---|---|
| 0 | Editor está aberto e nenhum documento está aberto. A única operação disponível para execução é open. |
| 1 | Após da entrada de dados do usuário ser equivalente à abertura de um documento, o sistema executa a operação open. |
| 2 | Neste estado, o documento está aberto. Os possíveis comandos são: |
| | 1: edit |
| | 2: print |
| | 4: close |
| 3 | Usuário escolheu fechar o documento. |
| 4 | Sistema executa a operação close. |
| 5 | Fim de execução. |
| 6 | Sistema executa a operação edit. |
| 7 | Neste estado, seria possível fechar o documento, porém os testes foram escritos de tal maneira a apenas verificar que as propriedades fossem obedecidas e neste modelo não é possível fechar o documento a partir deste estado.<br>Neste ponto do modelo, o documento foi editado. Os possíveis comandos são: |
| | 2: print |
| | 3: save |
| 8 | Sistema executa a operação save. |
| 9 | Sistema executa a operação print. |
| 10 | Sistema executa a operação print. |

Tabela 4.3 - Descrição dos estados e arcos do modelo 1

De acordo com os testes realizados no LTSA, o modelo 1 satisfaz as propriedades definidas. O documento é fechado após aberto, um documento somente é salvo após a operação de edição e nenhuma operação é realizada antes de abrir um documento.

Como acréscimo à especificação original, foi criada a operação *saveas*, na qual um documento pode ser salvo com um nome definido pelo usuário. As propriedades foram mantidas inalteradas para esta rodada de testes.

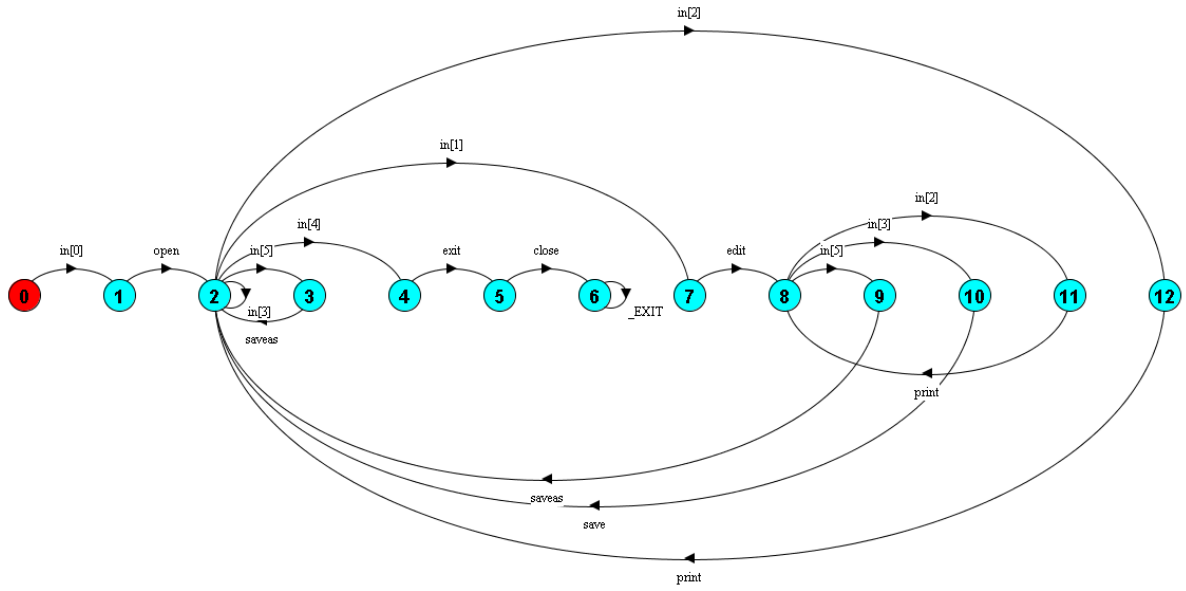


Figura 4.13: Modelo 2: Operação saveas  adicionada manualmente, sem alterar as propriedades.

Ao modificar o código original, de acordo com a nova especificação, um novo modelo foi gerado utilizando-se a mesma metodologia e ferramentas. Com a inserção de da operação saveas, o conjunto de testes foi alterado.

**T1: Sem alterações**

**T2: Sem alterações**

**T3: Operações em sequência**:

- Descrição: Testa o uso de todas as operações em sequência: open, edit, print, save, saveas, close.
- Vetor de inputs: [0, 1, 2, 3, 5, MeuArquivo.txt, 4]

- Resultado Esperado: Todas as operações são corretamente executadas na sequência chamada.

**T4: Operação saveas**:

- Descrição: Permite que a operação saveas possa ser executada em qualquer momento entre as operações open e close.
- Vetor de inputs: [0, 2, 1, 5, MeuArquivo.txt, 2, 5, MeuArquivo.txt, 3, 5, MeuArquivo.txt, 4]
- Resultado Esperado: Todas as operações são corretamente executadas na sequência chamada.

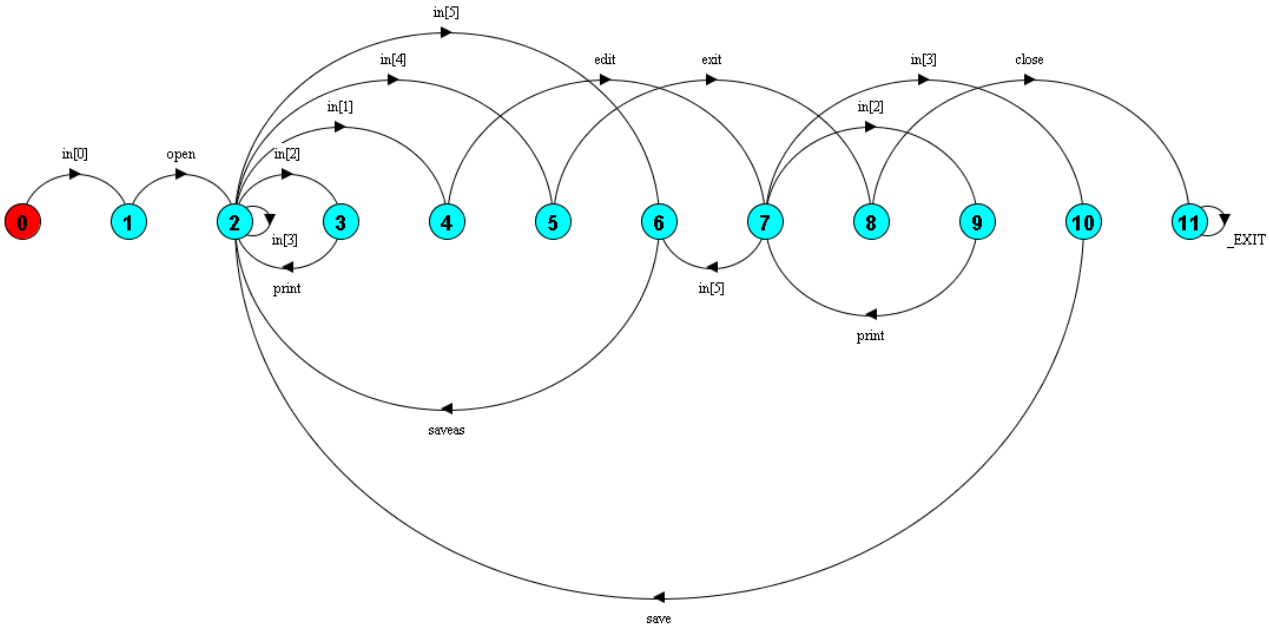


Figura 4.14: Modelo 3: Operação saveas adicionada no código. Modelo gerado de forma automática, sem alterar as propriedades.

Definindo o comportamento do modelo gerado manualmente como o correto, é preciso verificar se tal comportamento também ocorre no modelo gerado automaticamente. O modelo gerado através da validação do modelo automático, utilizando o modelo manual como propriedade é apresentado a seguir:

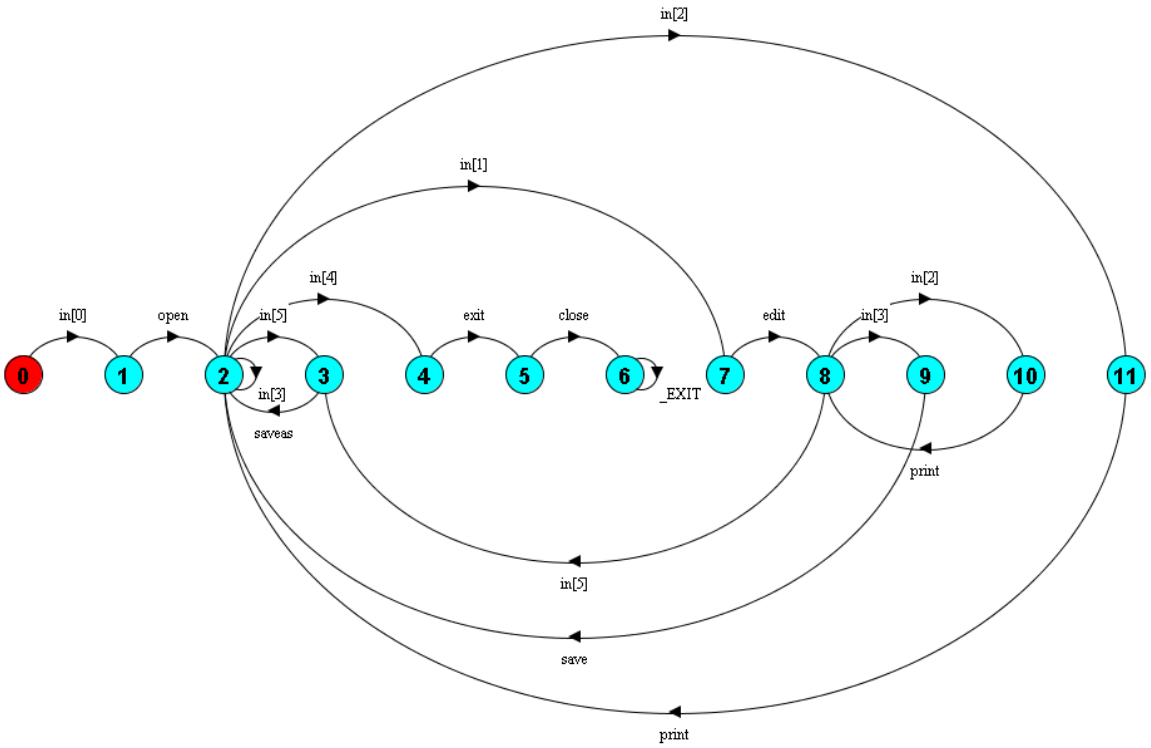


Figura 4.15 – Verificação entre os modelos 2 e 3.

Com a ausência de um estado de erro, verifica-se que o modelo gerado através do código reflete os comportamentos estabelecidos pelo modelo gerado manualmente.

Ampliando a especificação do Editor, adicionam-se duas novas funções: Undo e Redo. Como estas funções alteram o conteúdo do arquivo, somente podem ser usadas logo após a função Edit. Também estabeleceu-se um limite de 3 undos e redos, além de só ser possível utilizar a função redo se a função undo for usada anteriormente pelo menos o número de vezes que a função redo será usada.

A especificação atualizada tem as seguintes propriedades:

Propriedade 1: Todo documento aberto deve ser fechado

property OpenAndClose = CLOSED,

CLOSED = (open -> OPEN),

OPEN = (close -> CLOSED).

Propriedade 2: Um documento pode ser salvo apenas após ser editado

property SaveOnlyIfEdited = SAVED,

SAVED = (edit -> EDITED),

EDITED = (edit -> EDITED

|save -> SAVED).

Propriedade 3: Um documento pode ser editado, impresso ou salvo apenas após ter sido aberto.

fluent Open = <open, close> initially 0

assert Allowed = (!Open -> !(edit || print || save))

Propriedade 4: O conteúdo de um documento só pode ser desfeito ou refeito enquanto estiver sendo editado.

Property UndoRedoOnlyIfEdited = UNDOREDO,

UNDOREDO = (edit ~> EDITED),

EDITED = (edit ~> EDITED

|undo ~> UNDONE

|redo ~> REDONE).

Propriedade 5: A função Redo pode ser utilizada somente após a função Undo ter sido utilizada ao menos uma vez.

Property RedoAfterUndo = (undo -> redo -> RedoAfterUndo).

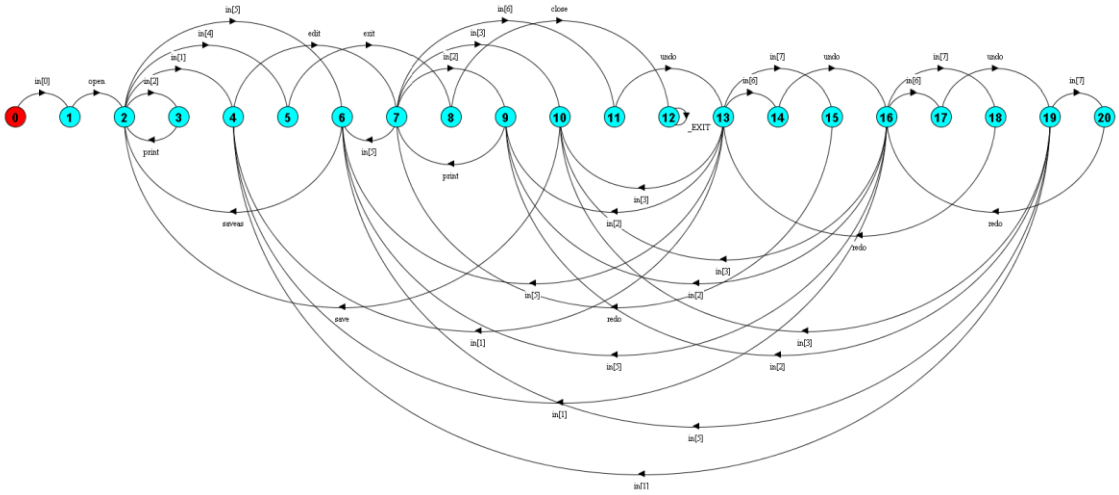


Figura 4.16: Modelo 4, modificado manualmente com Undo/Redo

A inserção de dois novos métodos com restrições explícitas gerou uma grande complexidade na alteração manual do modelo já existente. Foi preciso cuidado ao criar caminhos entre os estados que permitissem apenas que no máximo três operações undo ou redo fossem utilizadas, além de permitir que as funções previamente existentes continuassem a funcionar antes e após a utilização das novas funções. Já a alteração no código foi realizada rapidamente, de maneira simples e sem inspiração no modelo gerado automaticamente. Foram criados dois métodos e um contador para o número de "undos" realizados. Cada chamada do método undo incrementa o contador e cada chamada do método redo decrementa o contador. O método redo só pode ser executado se o contador countUndo possuir um valor maior que 0, assim é garantido que toda chamada de redo será feita após uma chamada de undo. Além disso, existe uma condição no código que só executa a o método undo caso o valor de countUndo seja menor que o inteiro 3. Por último, o valor do contador é alterado para o inteiro 0 caso as opções edit, print, save ou save as sejam utilizadas.

Código alterado:

```
countUndo = 0;

 void edit () {
   countUndo = 0;
   isSaved = false;
   System.out.println ("\n Edite");
 }

 void print () {
   countUndo = 0;
   System.out.println ("\n Print");
 }

 void save () {
   countUndo = 0;
```

```
  isSaved = true;
  System.out.println ("\n Save");
}

void saveas () {
        countUndo = 0;
     isSaved = true;
     System.out.print ("\n Type the document name:  ");
     try {
       String rc;
         rc = r.readCommand ();
         documentName = rc;
     }
     catch (Exception e) {}
     System.out.println ("\n Saveas");
}

void undo () {
       countUndo++;
     System.out.println ("\n Undo");
    }

void redo () {
       countUndo--;
     System.out.println ("\n Redo");
    }
```

O último passo realizado foi a alteração do conjunto de testes e extração do modelo. Para isso, foi necessário criar casos de teste para garantir as restrições estabelecidas ao uso de undo e redo, além de identificar se todas as funcionalidades do sistema funcionam corretamente após o uso de undo ou redo.

T5: “Happy Path” Undo/Redo

- Descrição: Teste para realizar o fluxo comum de execução das operações undo e redo. Testam-se todas as combinações de undo e redo com no máximo 3 repetições de cada operação.
- Vetor de inputs: [0, 1, 6, 7, 6, 6, 7, 7, 6, 6, 6, 7, 7, 7, 6 ,6 ,7 ,6, 6, 7, 4]
- Resultado Esperado: Após a chamada de edit, todas as operações de undo e redo são corretamente realizadas na ordem que aparecem no vetor de inputs, imprimindo os valores “undo” e “redo” na tela.

T6: Teste Negativo Undo/Redo.

- Descrição: Teste que verifica a chamada de uma função redo antes de uma função undo, além de testar mais de 3 chamadas consecutivas destas funções.
- Vetor de inputs: [0, 1, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 4]
- Resultado Esperado: A primeira chamada de redo não deve imprimir resultados na tela. Somente as 3 primeiras chamadas de undo e redo são efetivamente executadas.

T7: Teste de Regressão Undo/Redo

- Descrição: Teste que verifica a chamada de todas as funções já existentes no sistema após a chamada de undo ou redo.
- Vetor de inputs: [0, 1, 6, 2, 6, 3, 6, 5, MeuArquivo.txt, 1, 6, 6, , 2, 6, 6, 3, 6, 5, MeuArquivo.txt, 1, 6, 6, 6, 2, 6, 6, 6, 3, 6, 6, 6, 5, MeuArquivo.txt, 4]
- Resultado Esperado: Todas as operações imprimem valores válidos na tela.

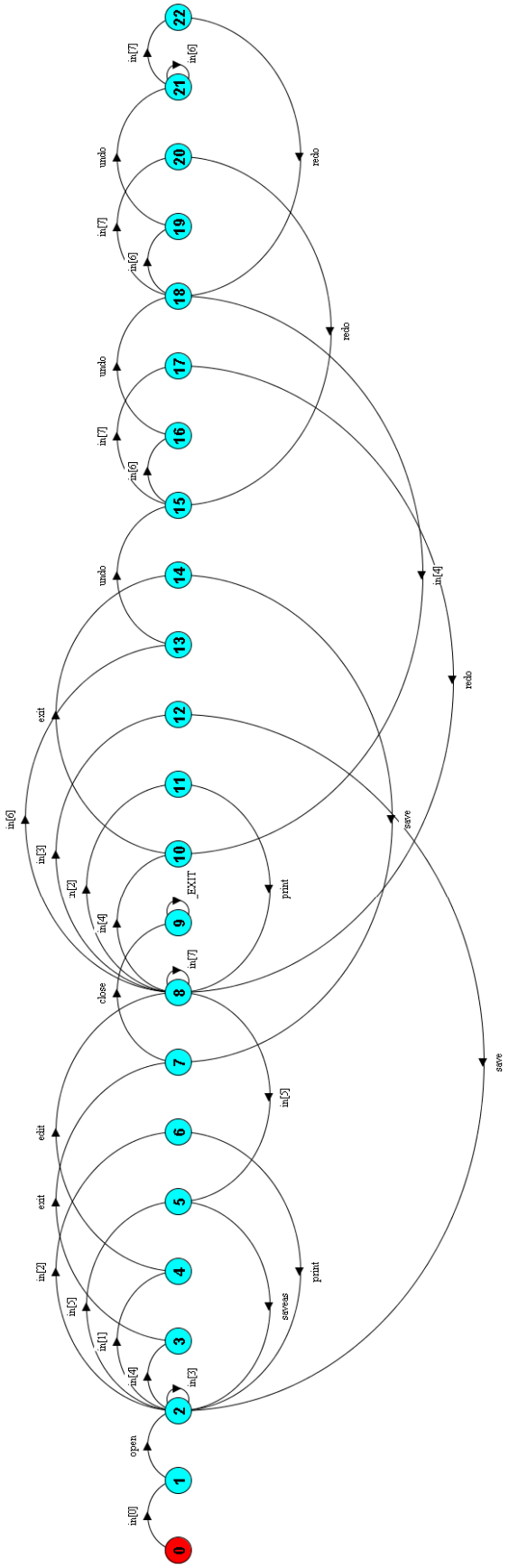


Figura 4.17: Modelo 5, extraído do código-fonte com Undo/Redo

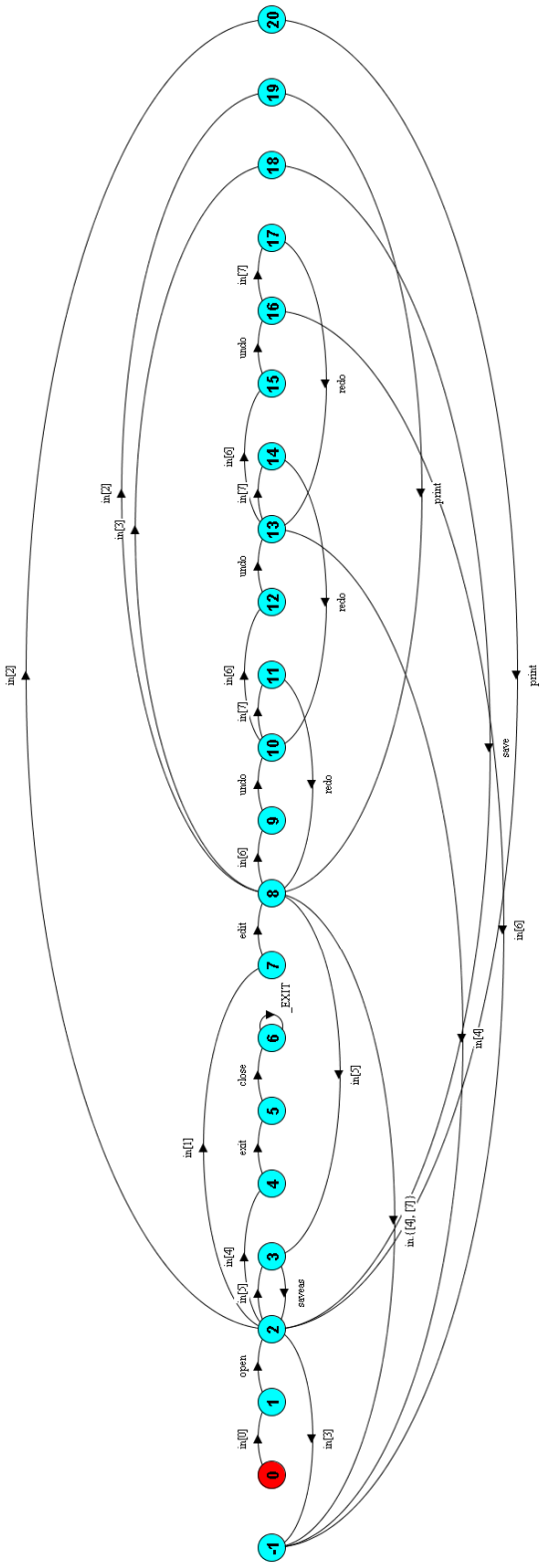


Figura 4.18: Verificação dos modelos 4 e 5.

Após a verificação, observa-se que o modelo resultante possui um estado de erro, o qual é o destino de quatro arcos. Apesar de este fato apontar diferenças entre os dois modelos verificados, o comportamento de ambos não aponta muitas diferenças. Todos os arcos pertencentes ao estado de erro representam valores de input do sistema que não estão ativando métodos durante a execução. Isto significa que a partir do estado de origem destes arcos, a função chamada através destes inputs não poderá ser executada, o que representa um comportamento correto para os valores 3, 6 e 7. O inteiro 4 é um valor de input para terminar a execução do sistema e segundo o código fonte, pode ser executado a qualquer momento em que um novo valor de input é lido. Sua ausência em outros estados deve-se a uma cobertura de testes incompleta e sua ausência no modelo gerado manualmente reflete erro humano de modelagem.

| Editor | | | |
|---|---|---|---|
| | Número de Estados | Número de Arcos | Número de Erros |
| Modelo 2 | 13 | 19 | 0 |
| Modelo 3 | 12 | 18 | 0 |
| Modelo 4 | 21 | 41 | 5 |
| Modelo 5 | 23 | 36 | 0 |

Tabela 4.4: Resultados dos experimentos realizados com Editor

Os erros ocorridos no modelo 4 correspondem a valores de input. Isto significa que o modelo manual não levou em consideração certos comportamentos do sistema os quais foram capturados pelo conjunto de testes. Tais comportamentos não ferem as propriedades descritas na especificação e fazem parte de uma cobertura de testes maior do que a cobertura mínima necessária para capturar os comportamentos esperados. Desconsiderando inputs e considerando apenas funcionalidades, o modelo apresenta comportamentos corretos.

Ao analisar cada experimento separado, chegou-se às seguintes conclusões:

Saveas: Assim como a funcionalidade print, não havia restrições quanto ao uso da funcionalidade saveas, sendo simples verificar tanto no modelo quanto no código, que alterações eram necessárias para se obter o comportamento esperado.

Undo/Redo: Dadas as restrições, onde apenas 3 chamadas de undo e redo seguidas poderiam ser feitas, além da necessidade de ser chamado um undo antes de um redo, a combinação de caminhos não é trivial de ser modelada manualmente. A alteração do modelo foi visivelmente mais trabalhosa, ao ser comparada tanto com alterações estudadas anteriormente, quanto ao código modificado do Editor. Para o código, bastou definir a chamada dos métodos caso algumas condições fossem respeitadas, tal qual é descrito na propriedade. A modelagem manual não foi realizada com sucesso na

primeira vez, sendo necessárias algumas revisões e testes de mesa para validar o comportamento o mais próximo possível do esperado.

Com o segundo experimento feito no Editor, o modelo gerado automaticamente foi mais compacto, legível e com menos erros, quando comparado ao modelo manual. Contrariando a hipótese inicial deste trabalho e os resultados obtidos até então, este experimento mostrou que podem existir tipos de alterações nas quais a modificação manual do modelo não é mais simples do que a alteração do código e geração automatizada do modelo.

# 5 CONCLUSÃO

Neste trabalho, estudaram-se diferentes tipos de modificações, com diferentes complexidades para alteração do modelo ou código. Assim como as modificações, os sistemas utilizados durante os experimentos não possuíam comportamentos complexos, sendo restritos a poucas funcionalidades e propriedades.

Observou-se que em todos os casos estudados, não há diferença significativa entre os modelos, tanto para número de estados, transições ou erros encontrados. Numericamente falando, é possível concluir que modelar comportamentos tanto manualmente a partir de um modelo existente, quanto gerar um modelo a partir do código alterado rendem resultados aceitáveis e semelhantes, para modificações simples ou de menor complexidade. Ao analisar os erros de cada modelo em específico, é notável que modelos gerados manualmente podem ser consertados de maneira mais simples quando o erro é encontrado, pois não há necessidade de alterar o código ou a suíte de testes para executar novamente os passos de extração. Tais passos e análises são necessários quando o erro é encontrado no modelo extraído.

A vantagem de possuir dois modelos potencialmente redundantes é vista quando as possibilidades são analisadas. Caso o modelo extraído do código seja correto, pode-se utilizá-lo como documentação estável e descartar o modelo manual, o qual possui sua função no passo de verificação de modelos. Caso o modelo manual seja o único correto, servirá de guia para que os erros encontrados no código possam ser corrigidos. Ambos os casos fizeram-se presentes durante os experimentos realizados.

Tanto a geração manual quanto a extração de modelos apresentaram resultados semelhantes, não mostrando maiores vantagens em um método sobre o outro. Porém é importante observar como a verificação de um modelo sobre o outro facilita a análise dos resultados e que esta verificação de um modelo sobre o outro, ao invés de um modelo sobre as propriedades da especificação é um passo importante do processo.

Para incrementar o processo estudado, seria interessante um estudo que analisasse diferentes tipos de modificações, as quais não entraram no escopo deste trabalho. Não foi estudado o impacto de alterações que excluam parte do comportamento existente, ou que modificam comportamentos, em vez de adicionar novos. Além disso, os sistemas estudados possuem funções simples, de fácil entendimento. Sistemas reais são complexos, e um estudo de sistemas com esta característica podem revelar resultados importantes do uso de extração e verificação de modelos durante o ciclo de vida dos mesmos.

# REFERÊNCIAS

PEZZÈ, M.; YOUNG, M. **"Teste e Análise de Software. processos, princípios e técnicas"**, Porto Alegre:Bookman,2008

MAGEE, J.; KRAMER, J. **"Concurrency. state models & java programs"**, Wiley, 1999

HOLZMANN, G.; Smith, M. **"A Practical Method for Verifying Event-Driven Software"**, in 'International Conference of Software Engineering', ACM New York, Los Angeles, USA, p. 507-607, 1999.

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

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **"Introdução à Teoria de Autômatos, Linguagens e Computação",** Ed. Campus, 2002

DUARTE, L.; KRAMER, J.; UCHITEL, S. Model Extraction Using Context Information**. Model Driven Engineering Languages and Systems**, Genova, p. 380-394, out. 2006