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

LUIZ FELIPE NOGUEIRA MAJERKOWSKI FILHO

Integração entre Verificação de Modelos e Teste de Software para Melhoria da Detecção de Erros em Sistemas Computacionais

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.

Prof. Dr. Lucio Mauro Duarte
Orientador

Prof. Dra. Érika Cota
Co-orientadora

Porto Alegre, julho de 2012.

AGRADECIMENTOS

Gostaria de agradecer primeiramente aos meus orientadores pelo suporte e dedicação exercida nas atividades de acompanhamento deste trabalho. Também gostaria de agradecer a minha família que me deu grande apoio e exemplo na vida. Por ultimo, mas não menos importante, a minha namorada que graças a sua presença me ajudou enormemente para conseguir completar este trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

LTS	Labelled Transition Systems
LTL	Linear Temporal Logic
LTSA	Labelled Transition System Analyzer
FLTL	Fluent Linear Temporal Logic
FSP	Finite State Process
LTSE	Labelled Transition System Extractor

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Os sistemas de computação estão presentes em diversas atividades de vital importância atualmente. Por este fato, a verificação do comportamento de aplicações é uma tarefa fundamental para se garantir a conformidade da execução das aplicações com seu comportamento esperado. Mesmo com técnicas formais, não é possível garantir completamente o comportamento correto de uma aplicação, mas consegue-se aumentar a confiança sobre a sua correta execução.

Este trabalho busca estabelecer um algoritmo que integre as técnicas de verificação de modelos e teste de software. Este algoritmo deve aumentar a cobertura dos testes e, ao mesmo tempo, melhorar a completude de um modelo abstrato do comportamento da aplicação. Desta forma, é possível encontrarem-se eventuais problemas com o código da aplicação através de um procedimento automático e que possa ser repetido onde, gradualmente, se melhora a cobertura oferecida pelas duas técnicas utilizadas.

Esta integração é feita através do uso de modelos de comportamento gerados partindo-se de um conjunto inicial de testes, através de uma técnica de extração de modelos. Após a geração de um modelo inicial, o modelo é verificado contra um conjunto de propriedades. Em caso de violações, novos casos de testes são gerados para garantir que os problemas encontrados são erros reais. Em caso de erros espúrios, o modelo é refinado para eliminar o comportamento inválido. Quando o modelo está correto, começa-se a gerar novos testes, baseados no modelo, para aumentar a cobertura de testes sobre a aplicação. Espera-se que durante este processo se identifiquem novos possíveis problemas na aplicação para que se possa corrigi-los.

Validou-se o algoritmo em estudos de caso, conseguindo-se executá-lo de forma parcialmente automática. Alguns problemas reais foram identificados e corrigidos. Adicionalmente, um conjunto de novos testes foi gerado e a completude do modelo foi aumentada, assim atingindo o objetivo do algoritmo.

Palavras-Chave: Verificação de modelos, Teste de Software, Extração de Modelos.

Integration of Model Checking and Software Testing to Improve Failure Detection in Software Systems

ABSTRACT

Software systems are present in almost every crucial daily activity nowadays. For this reason verifying the behavior of a given software system is fundamental to guarantee the conformance between the intended behavior and the actual execution of the given application. Even when formal methods are used, it is not possible to completely ensure the correct behavior of an application, however it is possible to increase the confidence on the correct behavior of that given application.

This work tries to establish an algorithm that integrates Model Checking and Software testing. This algorithm should increase the testing coverage and, at the same time, be able to increase the completeness of a given abstract behavior model of the application. This way, one can discover errors in the application through an automatic and reproducible method that can gradually increase the coverage provided by the applied techniques.

This integration is performed through behavior models generated using a model extraction approach base on a set of test cases. After generating the initial model, this model is checked against a set of properties of interest to ensure its correctness. When a violation occurs, a set of test cases is generated to check whether it is a real violation. If a spurious violation is detected, the model is refined to eliminate the invalid behavior. When the model is considered correct, new test cases are created based on that model in order to increase the testing coverage of the application. It is expected that during that process new errors are identified so that one can fix them.

The algorithm was validated through case studies and was executed in a semi-automatic way. A given set of real application errors was identified and fixed. In addition to that, a set of new test cases was produced increasing the testing coverage of the application, thus reaching the expected result for the algorithm.

Keywords: Model Checking, Software Testing, Model Extraction.

1 INTRODUÇÃO

1.1 Contexto

Atualmente, a verificação do comportamento de aplicações é uma tarefa fundamental para se garantir a conformidade da execução das aplicações com seu comportamento esperado. Mesmo com o uso de técnicas formais, não é possível garantir-se que a aplicação está totalmente correta, mas é possível aumentar o grau de confiança quando certos casos são verificados e se tem indicativos de que não ocorrerão (CLARKE, 1996).

Buscam-se então técnicas que ajudem na identificação de problemas e no aumento do grau de confiança sobre a aplicação. Dentre as diversas técnicas, podem-se ressaltar o teste de software e a verificação de modelos.

Uma abordagem de teste de software baseia-se na criação de vetores de teste buscando exercitar a ampliação a fim de encontrar discrepâncias em relação a uma especificação. Com uma abordagem de *teste de software* tem-se a vantagem de que todos os comportamentos identificados após a execução dos testes são comportamentos reais da aplicação, assim identificamos uma vantagem em relação à técnica de criação de modelos independentes de código, pois no modelo podem-se ter presentes informações que foram adicionadas durante a fase de criação do modelo e que não são comportamentos reais da aplicação (PRESSMAN, 2005). Assim, com as técnicas de teste software, tenta-se aumentar a cobertura de teste sobre o código e ajudar na identificação de problemas de forma não exaustiva. As principais desvantagens do uso do teste de software de uma forma isolada é a alta complexidade para gerar testes que identifiquem falhas de sobreposição ou execução não linear de ações e a baixa capacidade de automação da verificação da conformidade da aplicação contra uma especificação do comportamento esperado para o sistema.

Uma abordagem de verificação de modelos baseia-se na análise de um modelo do comportamento do sistema em relação a uma especificação em busca de discrepâncias. Como um modelo de comportamento é uma descrição abstrata do comportamento esperado de um sistema (UCHITEL, KRAMER e MAGEE, 2003), uma das vantagens é a possibilidade de automatização, visto que pode-se analisar o modelo mesmo em situações em que seria muito difícil trabalhar diretamente no código (CLARKE, GUMBERG e PELED 1999), por seu tamanho e complexidade. Os modelos podem ser utilizados como entrada para ferramentas que fazem a análise automática dos comportamentos contidos nos modelos (DUARTE, 2007). Destas análises, pode-se destacar a eficiência do uso de verificação de modelos para identificar possíveis ciclos e

violações de propriedades, as quais podem ser descritas em alguma Lógica temporal como, por exemplo, *Linear Temporal Logic* (LTL). (MANNA, PNUELI, 1992).

Por alguns dos problemas com aplicações ocorrerem mais especificamente quando existe a sobreposição ou a chamada não linear de ações, fica evidente necessidade da existência de um modelo que represente estes comportamentos e possa ser composto de forma não linear. Estes comportamentos não são trivialmente observáveis ou reproduzíveis utilizando apenas testes de software (GROZ, LI, PETRENKO, SHAHBAZ, 2008). Porém a construção do modelo se torna um fator importante quanto ao uso de técnicas baseadas em modelos. Esta construção não pode ser um fator custoso o suficiente que seja mais rápido executar as validações diretamente na aplicação em vez de gastar-se tempo gerando o modelo (HOLZMANN, 2001). Outro fator importante é a garantia da correção do modelo. Se sua geração for totalmente independente do código da aplicação, o modelo poderá não representar corretamente os comportamentos executados pela aplicação.

Desta forma, pode se notar uma oportunidade de integrar as técnicas buscando se valer das suas qualidades e mitigando seus pontos fracos. Como visto em (GROZ et al. 2007), (WALKINSHAW, DERRICK, GUO, 2009) e (BEYER et al., 2004) existem tentativas de integração entre técnicas de teste de software e verificação de modelos. Neste trabalho se tenta utilizar uma abordagem similar. Outro exemplo da tentativa de integração pode ser o uso de uma técnica de geração de testes baseados em contraexemplos contidos em modelos após execução de verificação, podem-se criar novos testes focados em identificar as classes de erros relativos ao contraexemplo (BEYER, CHLIPALA, HENZINGER, JHALA, MAJUMDAR, 2004).

1.2 Objetivo

O objetivo deste estudo é a criação de um algoritmo que integre as vantagens das abordagens de verificação de modelos e teste de software, também buscando mitigar as possíveis desvantagens identificadas. Uma vantagem da verificação de modelos, que se pode ressaltar, é a automatização do processo. Por outro lado, uma vantagem que se pode ressaltar do teste de software é a garantia de que comportamentos derivados do teste são considerados reais, pois se tem uma evidência deste comportamento após a execução do código.

É desejável que:

- A execução do algoritmo seja automática, assim diminuído a dependência de iteração humana na execução;

- Consiga-se aumentar a cobertura de teste da aplicação, partindo-se de um conjunto de testes inicial;

- Obtenha-se um modelo completo e correto em relação ao comportamento da aplicação considerando-se a especificação; e

- Consiga-se identificar erros na aplicação com a finalidade de se buscar a correção do programa.

As técnicas devem ser integradas de forma simples. Esta integração deve ser feita através da utilização dos modelos como base de representação das informações sobre a aplicação e o teste de software sendo responsável por adicionar informações pertinentes ao modelo e detectar erros não observáveis no modelo. A verificação de modelos deve ajudar na identificação de problemas no código através de indicativos de erros identificáveis no modelo que representa a aplicação.

O resultado esperado é que se identifique o maior número de problemas existentes na aplicação; que se consiga adicionar testes buscando o acréscimo de cobertura de testes ao código; que ao final da execução do algoritmo se tenha um modelo representativo da aplicação e que o algoritmo possa reduzir a quantidade de interação humana durante a fase de procura por problemas.

1.3 Metodologia

De forma similar a alguns trabalhos já executados nesta área (WALKINSHAW et al., 2009), (GROZ et al.,2008) e (BEYER et al., 2004), são utilizadas algumas ferramentas de apoio para a execução do algoritmo e para a geração e verificação de modelos, bem como se utilizam técnicas de teste de software para a geração de testes adicionais, partindo-se de informações contidas no modelo.

Inicialmente, é utilizada uma técnica de extração de modelos (DUARTE, 2007) para gerar-se um modelo inicial. Este modelo inicial é gerado partindo-se de um conjunto inicial de testes. Assim, como entradas para o algoritmo são necessários um conjunto mínimo de destes e uma especificação sobre a aplicação em alguma lógica formal.

Utiliza-se uma ferramenta de verificação e análise de modelos, que nos permite verificar se o modelo gerado viola as propriedades inicialmente propostas. Esta ferramenta também nos disponibiliza uma visualização gráfica do modelo gerado (MAGEE, KRAMER, 2006).

Durante a execução deste algoritmo se busca que o modelo esteja sempre correto de acordo com a especificação, aumentando sua representatividade a fim de identificar a maior parte dos comportamentos da aplicação. Adicionalmente, através do uso de teste de software, se tenta aumentar a cobertura de testes adicionando novos testes partindo do modelo.

Este processo é feito de forma gradual e incremental e a cada ciclo de execução busca-se aumentar a cobertura dos destes e a representatividade do modelo em relação a aplicação.

Após a execução do algoritmo, espera-se ter um conjunto de novos testes gerados partindo do modelo, um conjunto de ações identificadas como não executáveis pela aplicação e um conjunto de falhas identificadas na aplicação, que possivelmente foram corrigidas durante a execução do algoritmo, a fim de aumentar a confiança e diminuir o número de falhas na aplicação.

1.4 Estrutura do texto

Este trabalho está estruturado da seguinte forma:

No segundo capítulo, é feita uma revisão bibliográfica dos principais conceitos que foram utilizados como base para a elaboração do algoritmo proposto. No terceiro capítulo é feito uma descrição inicial do algoritmo de forma intuitiva e posteriormente a descrição das etapas e processos utilizados no algoritmo de maneira formal.

No quarto capítulo são apresentados os resultados da execução do algoritmo para alguns estudos de caso. Os resultados são demostrados para cada aplicação. São discutidos pontos importantes sobre o algoritmo proposto, como ponto de parada, limites, possibilidade de automação e custos.

O último capítulo apresenta a conclusão do trabalho e discutem-se os trabalhos futuros, possíveis limites e ganhos conseguidos com este trabalho.

2 REVISÃO BIBLIOGRÁFICA

Nesta seção, são apresentados os principais conceitos abordados e utilizados durante este trabalho. São descritas as características e especificações relativas à utilização de modelos, suas características principais, os tipos de modelos que são referenciados e as técnicas de verificação de modelos utilizadas neste trabalho. Também são abordados conceitos fundamentais sobre teste de software que servirão de base para o entendimento do algoritmo que será debatido. Por fim, tem-se uma discussão sobre a integração entre os conceitos de verificação de modelos, utilizando uma modelagem comportamental representada por um autômato de estados finitos, e as técnicas de teste de software, com foco na geração de testes baseada em modelos.

2.1 Verificação de Modelos

Podemos definir a *verificação de modelos* como o processo automático em que um conjunto de propriedades é verificado contra um modelo, que representa a aplicação, com o objetivo de checar se estas propriedades são satisfeitas. Segundo (CLARKE et al.,1996), a verificação de modelos pode ser divida em três fases distintas, a saber: modelagem, especificação e verificação.

A *modelagem* deve ser feita de forma a construir uma representação que possa mapear os comportamentos relevantes da aplicação-alvo. Esta modelagem deve ser feita utilizando um formalismo que seja compatível com a ferramenta de verificação que seja usada. Na fase de *especificação* são definidas, em uma lógica formal, as propriedades ou comportamentos que esperamos que uma determinada aplicação possua. (LEUSCHEL et al.,2001) e (MANNA et al.,1992). A *verificação* do modelo consiste em analisar se o modelo criado satisfaz as propriedades definidas. Esta avaliação é automatizada por ferramentas chamadas de *verificadores*. Após a verificação ser executada, dois resultados podem ser observados: a especificação pode ter sido *satisfeita* ou podemos verificar uma violação de propriedade. Caso a especificação tenha sido satisfeita, então o modelo não apresenta nenhum comportamento que viole as propriedades definidas. Por outro lado, caso exista uma violação, o verificador usualmente exibe um *contraexemplo*; isto é, uma sequência de transições no modelo que representa um comportamento que gera violação da especificação (CLARKE et al., 1999).

É importante frisar que, como o modelo da aplicação é apenas uma abstração dos comportamentos do sistema, ele pode não incluir todos os comportamentos possíveis. Desta forma, quando um modelo for verificado e nenhum contraexemplo for identificado, pode-se concluir apenas que aquele modelo do sistema está correto em relação à especificação inicial.

2.1.1 Modelos

Modelos de comportamento são representações de comportamentos presentes em uma determinada aplicação. Neste estudo, são utilizados modelos do tipo "*Labeled Transition Systems*" ou LTS. Modelos deste tipo são um formalismo derivado de maquinas de estados finitos. Um modelo deste tipo utiliza suas transições para representar comportamentos atômicos que levam a aplicação a trocar de estado. Os estados representam contextos em que o sistema está esperando que uma transição ocorra para que exista uma mudança de estado. Formalmente:

> Um modelo LTS é da forma M =(S, si,∑, T) onde:
>
> - S é um conjunto finito de estados
> - $s_i \in S$ e representa o estado inicial
> - ∑ é um alfabeto (conjunto de nomes de ações) e
> - $T \subseteq S \times \sum \times S$ é uma relação de transição
>
> As transições são etiquetadas com os nomes de ações que fazem o modelo executar uma transição entre estados. Assim, dados dois estados s_0, $s_1 \in S$ e uma ação $a \in \sum$, então uma transição $s_0 \xrightarrow{a} s_1$ significa que é possível chegar ao estado s_1 partindo do estado s_0 através da ação "a". Desta forma, uma transição só ocorre se a ação "a" ocorrer (DUARTE, 2007).

A figura 2.1.1 apresenta um exemplo de modelo LTS que representa um programa onde existe a abertura e o fechamento de um arquivo. Este modelo representa a transição de um programa do estado inicial "0" para o estado "1" através da ação "*exit*". O modelo representa uma aplicação onde, partindo de um estado inicial e se executando a ação *"exit"*, chega-se a um novo estado onde temos a possibilidade de executar a ação *"close"*. Adicionalmente, pode-se executar no estado inicial do sistema a ação *"open"*, tantas vezes quanto necessário, enquanto o sistema se mantiver no estado inicial.

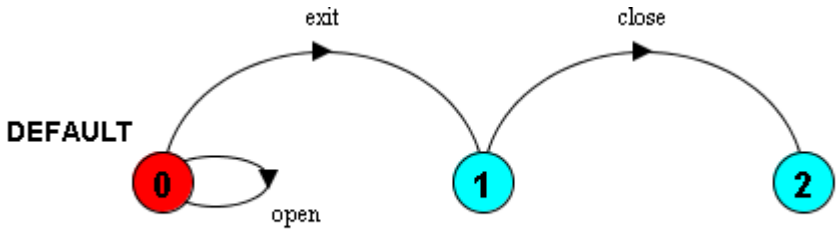

Figura 2.1.1: Exemplo de modelo LTS representando comportamentos de controle de abertura e fechamento de arquivo.

2.1.2 Refinamento do modelo

Durante a verificação do modelo contra as propriedades podem-se gerar *falsos negativos* (DUARTE, 2007). Estes falsos negativos são comportamentos existentes no modelo que não são realmente comportamentos verificáveis na aplicação. Por ser uma abstração, o modelo pode não conter informações suficientes para que uma propriedade seja verificada, assim gerando um falso negativo. Busca-se então uma forma de

aumentarmos a representatividade do modelo em relação à aplicação, utilizam-se então refinamentos no modelo.

O refinamento de um modelo é um processo que ocorre após a verificação do modelo. A ideia de refinar o modelo consiste em se adicionar informação ao modelo para que se consiga remover a presença daquele comportamento que pode ser considerado uma violação apenas do modelo e não da aplicação. Estas ambuiguidades são removidas através do processo de refinamento, quando estados que antes eram considerados iguais no modelo passam a representar situações diferentes pelo acréscimo de alguma informação nova.

Como visto em (DUARTE, 2007), pode-se definir uma relação de refinamento que além de remover os comportamentos do modelo não existentes na aplicação, garante também a preservação das propriedades do modelo. Nesta abordagem, o refinamento se dá pela utilização de valores de atributos dentro da representação dos estados de forma a definir novas classes de equivalência de estados.

2.1.3 Extração de modelos

A extração de modelos é um método de geração de modelo automático baseado em uma implementação já existente (HOLZMANN, 2001). Por ser uma tarefa automática, possui a característica de diminuir a interferência humana durante a criação do modelo.

Existem inúmeras propostas de extração de modelos, dentre elas podemos identificar os seguintes tipos: estática, dinâmica e híbrida (DUARTE, 2007). Pode-se caracterizar a abordagem *estática* pelo uso de informações contidas no código fonte do programa ou por informações contidas em diagramas de analise de controle de fluxo. (AHO, SETHI, ULLMAN, 1986). Pode-se caracterizar a abordagem *dinâmica* pela construção do modelo utilizando informações de execuções da aplicação para inferir padrões. Estes padrões podem ser gerados utilizando *inferência de gramáticas* (COOK, WOLF, 1998) ou *aprendizado de máquina (*AMMONS, BODIKÌK, LARUS, 2002*).*

Uma abordagem *hibrida* busca utilizar-se das informações estáticas coletadas do código e da analise do diagrama de fluxo de controle em conjunto com as informações retiradas da execução da aplicação coletadas de uma forma dinâmica para a geração de um modelo sobre a aplicação (ERNST, 2003).

Neste trabalho, utiliza-se uma abordagem híbrida onde, partindo-se de informações coletadas durante a execução da aplicação, gera-se um modelo LTS utilizando-se uma ferramentas de extração de modelos chamada *LTS Extractor* (LTSE), proposta em (DUARTE, 2007). A porção estática se utiliza de informações referentes ao fluxo de controle de execução da aplicação para auxiliar na geração do modelo. Adicionalmente, com uma abordagem dinâmica, utilizando-se de informações referentes a traços de execução, que são informações reais da execução da aplicação, e informações referentes às variáveis do programa, que são coletadas durante a execução, para criar informações sobre os "contextos" de execução e gerar o modelo.

Resumidamente, um *contexto* (DUARTE, 2007) é definido através da avaliação dos seguintes conceitos: o bloco corrente de código em execução, o qual é determinado pela análise dos predicados de controle; e os valores de atributos que definem um estado da aplicação. A definição formal de contextos é como segue: Dado um programa P, um contexto $C = (bc, val(pc), v)$ é a combinação em um certo ponto da execução do programa P do bloco corrente de controle denotado por *bc*, do valor da avaliação do

predicado de controle neste bloco denotado por *val(pc),* e o conjunto de valores de atributos pertencentes ao programa P denotados por *v*. Assim, pode-se definir a execução de um programa como um conjunto de diferentes contextos executados em alguma determinada sequência. A partir destas informações de contexto, podem-se determinar quais comportamentos são atingíveis pela execução do programa.

2.1.4 Correção e Completude de Modelos de Comportamento

A *correção* é um conceito muito importante para a verificação de modelos. No caso do modelo violar uma propriedade, mas a aplicação não violar esta propriedades, tem-se um caso em que o modelo exibe um comportamento não existente na aplicação. Neste caso, se diz que o modelo está *incorreto*.

A definição de Correção que será utilizada é a mesma discutida por Duarte (2007). Assumindo que temos um modelo M que está representando os comportamentos existentes no programa P. Os comportamentos existentes no programa serão representados por A(P) e os comportamentos existentes no modelo serão representados por A(M). O modelo M será considerado correto em relação ao programa P se e somente se A(M) $\subseteq$ A(P). Assim, quando estamos fazendo um refinamento no modelo, visando incluir mais informações, estamos tentando aumentar a correção do modelo removendo comportamentos que não são executáveis pelo programa em questão.

A *completude* também é um conceito de grande importância para a verificação de modelos. No caso da aplicação executar uma ação que viola as propriedades, porém o modelo não contiver aquele comportamento, tem-se um caso onde se diz que o modelo está *incompleto*. Assim, a completude se refere ao nível de representatividade que um dado modelo possui em relação a todos os comportamentos possíveis de uma determinada aplicação.

A definição de completude que será utilizada também foi discutida por Duarte (2007). Assumindo que temos um modelo M que esta representando os comportamentos existentes em um programa P. Os comportamentos existentes no programa serão denotados por A(P) e os comportamentos existentes no modelo serão denotados por A(M). O modelo será considerado completo em relação ao programa se e somente se A(P) $\subseteq$ A(M). Assim, a completude representa a propriedade em que buscamos que o maior número de comportamentos existentes na aplicação esteja presente no modelo.

2.1.5 Propriedades e especificação

As propriedades são características desejadas em determinada aplicação. Estas propriedades podem ser formalmente definidas através de uma lógica temporal que deve representa-las de forma não ambígua.

Dentre as diversas lógicas que existem, uma das mais utilizadas é a *lógica temporal linear*, ou LTL. A seguir podemos ver uma tabela dos operadores lógicos e temporais do formalismo utilizado para especificarmos propriedades. A semântica destes operadores pode ser encontrada em (LEUSCHEL et al.,2001) e (MANNA et al.,1992).

Assume-se, neste trabalho, que as propriedades são o oráculo que utiliza-se como base para verificação da correção da aplicação.

Como extensão ao formalismo *LTL*, pode-se utilizar o conceito de fluentes que foi introduzido por (GIANNAKOPOULOU, MAGEE, 2003). Um *fluente* constitui um atributo cujo valor verdade varia de acordo com a ocorrência de um conjunto de ações.

A extensão de LTL com o uso de fluentes é denominada Fluent Linear Temporal Logic (*FLTL)* (GIANNAKOPOULOU et al., 2003).

Tabela 2.1.1: Operadores Lógicos e Operadores temporais da lógica

Operadores Lógicos	Operadores Temporais
¬ (Negação lógica)	□ (sempre)
∧ (E lógico)	♦ (finalmente)
∨ (OU lógico)	○ (próximo)
⇒ (implicação)	*U* (until)
⇔ (equivalência)	*W* (until fraco)

Fonte: (DUARTE, 2007)

Considerando um alfabeto de ações A, temos: $F_l \equiv <I_{Fli}, T_{fli}>$, onde I_{Fl}, $T_{Fl} \subset A$ e $I_{Fl} \cap T_{Fl} = \varnothing$", ou seja F_l é um conjunto de I_{Fl} e T_{Fl} que são ações que mudam o estado corrente do fluente. Assim, o fluente será verdadeiro quando for inicializado com "Verdadeiro" ou se a ação $A \in I_{fi}$ ocorrer, ou seja, I_{fi} pode ser chamda de ação inicializadora. O fluente será falso se for inicializado com "Falso" ou se a ação $B \in T_{fi}$ ocorrer (GIANNAKOPOULOU et al., 2003), ou seja, T_{fi} pode ser chamada de ação terminadora. Por exemplo, *"fluent DOOR_CLOSED = <doorClosed,doorOpen> initially True"*, determina que o fluente *DOOR_CLOSED* é inicialmente verdadeiro até que a ação "*doorOpen"* ocorra e passará a ser falso. Caso a ação "*doorClosed"* ocorra o fluente volta a ser verdadeiro.

2.2 Teste de Software

Teste de software é a prática onde se criam ou se executam tarefas com o objetivo principal de tentar identificar o maior número possível de inconformidades entre um código e uma especificação. Teste de software, hoje em dia, é algo crítico para garantir a qualidade e é responsável pela última revisão feita sobre o código, à especificação e o desenho (PRESSMAN, 2005).

A técnica de testes baseia-se no fato de que, o teste de software de forma exaustiva não é possível para aplicações reais, onde o tamanho, custo e o tempo tendem a ser fatores limitantes. Assim, buscam-se executar o maior número de verificações dentro das restrições existentes sobre as três variáveis levantadas.

2.2.1 Teste funcional

O teste funcional é um tipo de teste em que se procura executar as funções definidas pela especificação ou de conhecimento do testador. Este tipo de teste visa executar a aplicação de forma a garantir e verificar que os comportamentos esperados estão de acordo com a especificação (PRESSMAN, 2005).

Como proposto por Pressman (2005):

> Quando utilizamos a técnica de teste funcional, devemos visar atingir os seguintes critérios: (1) criar casos de teste que busquem diminuir o número de casos de teste totais, pelo menos em um número maior que um e (2) criar testes que nos digam informações sobre a presença ou a ausência de classes de erros, ao invés de erros específicos (PRESSMAN, 2005).

Pode-se notar que a definição de teste funcional descreve um processo que busca identificar problemas com um determinado programa computacional. A técnica funcional busca achar casos de teste que consigam identificar múltiplos problemas e que diminua o número de testes exaustivos a serem executados no programa computacional.

2.2.2 Classes de equivalência

Classes de equivalência são abstrações de dados usualmente utilizadas para descrever conjuntos que são similares. Estes conjuntos são definidos com a finalidade de identificarem-se agrupamentos, onde um intervalo de valores possui a mesma resposta do sistema independentemente do valor escolhido dentro daquele intervalo.

Como definido por (ISTQB, 2011), classes de equivalência são grupos em que se espera um comportamento equivalente da aplicação e que sejam processados da mesma forma. Quando se utilizam valores presentes na mesma classe de equivalência, pode-se dizer que a aplicação deve executar exatamente as mesmas ações e ter os mesmos comportamentos.

Podem-se definir classes de equivalência sobre valores de entrada, valores de saída, valores de controle interno da aplicação dentre outros. As classes de equivalência também ajudam a reduzir o número de diferentes testes que devemos gerar para verificar uma determinada classe de erros. Por exemplo, uma única entrada inválida, que representa aquele grupo, é utilizada para validar a execução de um tratamento de exceção no código. Assim não se precisam testar as inúmeras possibilidades de entradas inválidas, pois todas seriam tratadas pelo mesmo caso de tratamento de exceção.

2.2.3 Teste baseado em Modelo

Testes baseados em modelos, atualmente, é uma área de grande importância e com inúmeras pesquisas como visto em (BERTOLINO, 2007), (WALKINSHAW, 2009) e (GROZ et al., 2008). A ideia deste tipo de teste é, partindo-se de um modelo desenvolvido para representar a aplicação, gera-se novos testes com a finalidade de aumentar a cobertura e identificar possíveis falhas decorrente da execução dos testes gerados na aplicação.

Pela complexidade das aplicações estarem crescendo em um ritmo acelerado, o teste manual consome um tempo excessivamente grande durante o ciclo de desenvolvimento. Por este fator, verifica-se a importância de termos uma forma de automatizar o processo de teste e diminuir o esforço manual para fazer geração e validação das aplicações (BERTOLINO, 2007). A estratégia utilizada é gerar teste de forma automática, baseados no modelo que representa a aplicação, e que consigam identificar comportamentos corretos, mas não existentes ainda no modelo. Também se pode ter o objetivo de gerar novos testes, partindo do modelo, que aumentem a cobertura de testes ou que encontrem falhas. Como visto em (GARGANTINI, 2007), umas das técnicas existentes é gerar testes partindo-se de contraexemplos, que são informações geradas após a verificação do modelo com alguma ferramenta automática. Na abordagem discutida por Gargantini (2007), gera-se, partindo de uma máquina de estados finitos, um conjunto de testes que irá revelar classes de erros na aplicação representada por este

modelo. Um dos problemas, inerentes dessa abordagem, é a forma como iremos gerar o modelo inicial e quais os critérios de parada para os ciclos de testes que utilizam modelos.

2.3 Verificação e Teste

Nesta seção do documento, iremos abordar algumas tentativas de utilização de verificação de modelos e teste de software de forma conjunta.

2.3.1 Integração de teste e verificação

Como discutido por (GARGANTINI, 2007), em sua abordagem, é feita uma integração entre a ferramenta de validação SPIN e a geração automática de teste, partindo de contraexemplos, que conseguem identificar, de forma automática, possíveis falhas na aplicação. O algoritmo, proposto por ele, usa contraexemplos gerados pela verificação do modelo para gerar conjuntos de testes, baseados em rastros de execução da aplicação transformados em axiomas, que consigam comprovar aquela falha na aplicação. Depois de gerado o teste, este deve ser executado na aplicação a fim de comprovar a validade do contraexemplo gerado. Assim comprovando a existência do erro encontrado no modelo na aplicação.

Utilizando outra abordagem, (GROZ et al., 2008) tenta utilizar um modelo baseado em entradas e saídas para derivar um modelo partindo de um conjunto de teste. Esta abordagem, porém, não define o número de testes necessário para inferir o modelo. No trabalho, são abordados refinamentos do modelo, porém também não existe uma ideia clara de quando ou o que fazer quando não é possível refinar o modelo.

Mas como se pode ver, em ambos os trabalhos a união das características da verificação de modelos e do teste se fazem importantes para os algoritmos propostos. Esta integração busca utilizar as qualidades inerentes de cada um dos dois métodos, como por exemplo, verificação de problemas assíncronos, tarefa facilmente feita com verificação de modelos como descrito em (GROZ et al., 2008). Também se pode notar a importância do teste de software na geração de traços de execução, que são utilizados para se aumentar a correção e a completude do modelo. Esta informação é por assim dizer real, tendo em vista que é resultado da execução da aplicação, assim diminuindo o problema do modelo não representar a aplicação de forma correta e completa (BEYER et al., 2004).

2.3.2 Geração de Testes partindo de contraexemplo

A técnica de geração de testes partindo de contraexemplos consiste em: após verificar um modelo M contra um conjunto de propriedades P e a verificação gerar um contraexemplo, utiliza-se o conjunto de ações ou conjunto de entradas e saídas que levam ao contraexemplo para gerar um teste na aplicação, buscando verificar se aquela violação do modelo é uma violação real da aplicação ou uma violação somente no modelo.

Como descrito por (BEYER et al., 2004) e (GARGANTI, 2007) a geração do teste baseado no contraexemplo é feita através do uso da sequência de ações que leva até o contraexemplo, gerado pelo verificador, de um modelo que represente a aplicação. Assim, com aquele conjunto de ações, se busca um vetor de entrada que resulte na

sequência de ações necessária para executar-se, na aplicação, a mesma sequência de ações contida no contraexemplo. Desta forma se buscando confirmar a existência daquele contraexemplo na aplicação.

3 ABORDAGEM PROPOSTA

A abordagem proposta utiliza as técnicas de verificação de modelos, extração de modelos e geração de testes baseada em modelos. Um algoritmo é utilizado para que se consiga, de forma semiautomática, integrar e aproveitar as vantagens das técnicas mencionadas. Para elaborar e descrever o comportamento do algoritmo se discutem as decisões e as heurísticas utilizadas para a sua criação**.** A elaboração do algoritmo foi baseada no estudo de propostas existentes que se utilizam, de forma independente, dos conceitos previamente citados. Neste estudo, diferentemente, busca-se uma integração entre verificação de modelos e teste de software.

Inicialmente, partindo-se de um conjunto mínimo de testes gerado de forma pseudoaleatória, ou seja, partindo de uma propriedade específica e buscando incluir uma sequência de ações contida na propriedade, é extraído um modelo inicial a partir da execução dos testes mínimos. O modelo é representado por um autômato de estados finito do tipo LTS, onde as transições representam ações representativas na aplicação-alvo que o fazem mudar de estado. Com o modelo inicial definido, tenta-se garantir sua correção e completude através do uso de verificação de modelos utilizando a ferramenta LTSA (MAGEE, KRAMER, 2006). Esta ferramenta verifica o modelo contra um conjunto de propriedades definidas em *Fluent Linear Temporal Logic* (FLTL) (GIANNAKOPOULOU et al., 2003) que representam os requisitos do sistema.

Quando o modelo inicial satisfaz as propriedades e não apresenta mais violações de propriedades, passa-se a tentar gerar novos testes baseados no modelo. Os novos testes são gerados por contraexemplos existentes no modelo quando este é transformado em propriedade. Estes contraexemplos surgem do fato de que todas as ações não existentes no modelo são consideradas violações. Assume-se que o que é complementar ao modelo não pode ser executado na aplicação, assim sendo designado como contraexemplo. Transformando-se o modelo em propriedade, todas as transições complementares ao modelo atual são inseridas no modelo e consideradas violações, pois não se tem informação da existência ou não destas transições. Estas transições adicionadas são chamadas de “transições negativas”, pois a ferramenta LTSA representa as violações ou contraexemplos como setas negativas com origem em um estado do modelo e destino em um estado de erro. Assim, busca-se identificar como falha real um contraexemplo. Gera-se um vetor de testes, utilizando classes de equivalência, com a finalidade de verificar se podemos executar, na aplicação, o contraexemplo. Neste estudo, chama-se de “transição negativa” o resultado da transformação do modelo em propriedade e cada transição complementar ao modelo é assumida como não executável. Assim, todas estas transições são representadas como violações e, na representação gráfica da ferramenta LTSA, são exibidas como transições para um estado de erro “-1”.

Definiram-se dois critérios para parada do algoritmo. Um critério para quando todos os testes adicionais foram gerados e não é possível encontrar mais falhas e outro para quando uma propriedade ou uma característica da aplicação extrapola a capacidade de representação do modelo utilizado.

Ao final da execução do algoritmo, espera-se que exista uma lista resultante de testes adicionais criados, uma lista de falhas corrigidas, uma lista de comportamentos não executáveis no modelo e um modelo final que representa a aplicação de uma forma abstrata.

A seguir, o algoritmo é apresentado em mais detalhes.

3.1 Heurística do algoritmo

Inicialmente, a ideia intuitiva do algoritmo é: partindo-se de uma forma automática de geração de modelos e com uma ferramenta de auxílio para verificação de propriedades, seriamos capazes de garantir a correção do modelo. Partindo deste modelo correto, consege-se gerar novos testes para aumentarmos a cobertura de caminhos e encontrar possíveis falhas não contidas no modelo inicial, mas existentes na aplicação.

O primeiro desafio encontrado durante a elaboração do algoritmo foi definir o conjunto de testes iniciais. Utilizando-se de algumas propostas estudadas por (GROZ et al.,2008), (GARGANTINI,2007) e (WALKINSHAW, 2009), a ideia seria estabelecer um conjunto mínimo de destes para se começar a execução dos testes. Foi definido que seria necessário apenas um teste inicial que contivesse ao menos alguma das ações existentes na especificação. Durante a execução do algoritmo, posteriormente, conseguiu-se verificar que a fase inicial de garantia de correção e completude acaba convergindo mais rapidamente quando o teste inicial inclui todas as ações que pertencem ao alfabeto da propriedade. Porém esta constatação não invalida a ideia de iniciar o algoritmo com um conjunto mínimo de testes, pois esta ideia não acarreta erros no processo, só uma possível demora na convergência.

Partindo-se do conjunto inicial de testes, instrumenta-se o código. Este passo consiste em adicionar instruções ao código, a partir de uma gramática de substituição, que mantenham a semântica do mesmo, mas possibilitem a extração de informações de contexto e das chamadas de métodos (DUARTE, 2007).

Após executar o conjunto de testes iniciais, com o código instrumentado, utilizam-se os traços da execução, ou seja, informações de contexto e chamadas de métodos, para gerar o modelo inicial conforme método descrito por Duarte (2007), utilizando uma ferramenta automática de extração de modelos chamada *LTS Extractor* (LTSE).

O modelo inicial segue uma ideia de modelo mínimo, que tenta atender a todas as propriedades propostas. Nesta fase, já é possível identificarem-se falhas na aplicação dependendo do conjunto de testes iniciais. Este modelo inicial é verificado contra as propriedades buscando encontrar violações. As violações podem ser de dois tipos: uma *violação real*, onde a aplicação realmente permite que a sequência de ações seja executada, ou uma *violação espúria*, onde podemos executar algo no modelo e não na aplicação por algum erro na modelagem. Um erro espúrio pode acontecer por dois fatores: a representação do modelo não possui informações suficientes sobre a

aplicação, possuindo menos comportamentos que a aplicação realmente possui - chamaremos de um erro de completude - ou a representação do modelo permite algo que a aplicação não executa - chamaremos de erro de correção. O algoritmo tenta corrigir os erros espúrios refinando o modelo, através de refinamento utilizando atributos da aplicação ou adicionando testes para aumentarmos a completude do modelo, visando a que o alfabeto do modelo seja igual ao alfabeto das propriedades.

Quando um modelo que não possui mais violações é encontrado, passa-se a tentar descobrir novos comportamentos através da geração de novos testes baseados no modelo atual. Nesta fase, busca-se aumentar a cobertura do modelo adicionando-se novos possíveis comportamentos ou sequência de ações que sejam permitidas na aplicação. Após a adição destas sequências de ações ao modelo inicial, as propriedades devem ser verificadas novamente para garantir que não foram violadas, mantendo, assim, a correção do modelo mesmo com o acréscimo de ações, e aumentando a sua completude. Neste passo, é possível que sejam encontradas falhas na aplicação, pois ao encontrar-se um novo comportamento ou sequência de ações que seja executável na aplicação e que fira uma propriedade, se tem uma falha encontrada, pois a violação realmente ocorre na aplicação.

A geração dos novos testes, para tentar verificar a existência de "*comportamentos não observados*", é feita partindo-se das "transições negativas" que surgem da transformação do modelo estável em propriedade. Estas transições negativas são a forma da ferramenta LTSA (MAGEE, KRAMER, CHATLEY, UCHITEL, FOSTER, 2012) identificar que uma ação não contida no modelo não pode ser executada, então é criada uma transição negativa do estado atual para um estado de erro. Como podemos ver na figura 3.1.1, as transições levadas para o estado de erro (representado por -1) são consideradas violações ou desconhecidas. Estas transições foram geradas a partir da transformação do modelo em propriedade e as ações não são permitidas no modelo original, assim são consideradas violações.

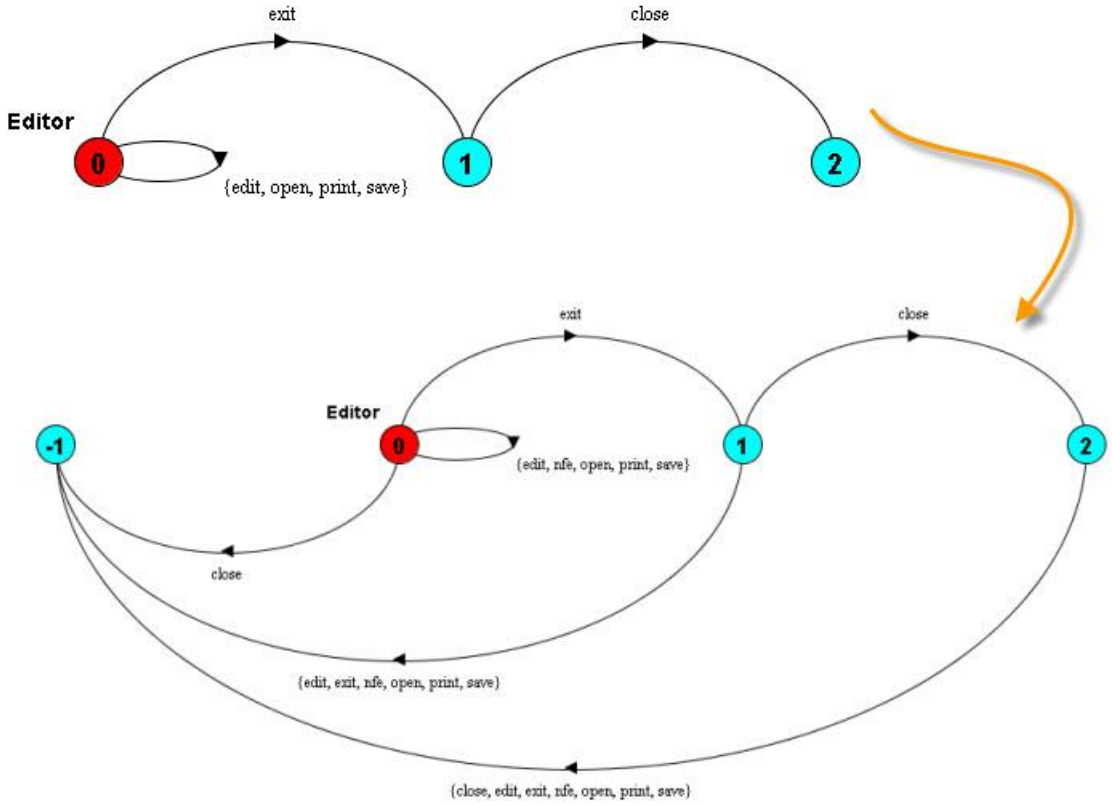

Figura 3.1.1: Exemplo de transição negativa gerada a partir da transformação do modelo inicial (parte de cima da figura) em propriedade (modelo na parte inferior).

Partindo-se das transições negativas, tenta-se achar uma classe de equivalência para gerar um vetor de teste que consiga executar a sequência de ações que leva até o estado de erro, ou seja, comprovar que aquela transição negativa é executável na aplicação. Caso alguma sequência não seja executável, guarda-se esta em uma lista que é utilizada para verificar-se quando se pode parar o algoritmo.

Ao final, após possíveis correções na aplicação e a geração de todos os novos testes, visando a eliminar a possibilidade de existirem sequências no modelo que não foram testadas, consegue-se chegar a um modelo final que representará a aplicação de forma correta e completa, considerando-se o nível de abstração do modelo. Teremos uma lista de defeitos encontrados, um conjunto de testes gerados a partir do modelo e um conjunto de sequências que se garante não serem executáveis na aplicação.

3.2 Algoritmo proposto

O algoritmo proposto segue a sequência de execução demonstrada no fluxograma da figura 3.2.1.

Este algoritmo foi desenvolvido de forma incremental. Inicialmente utilizou-se uma aplicação com pequena complexidade para se desenvolver uma heurística para o

algoritmo. Após alguns resultados relevantes terem sido encontrados, se estabeleceu o algoritmo e buscou-se executá-lo em um conjunto de estudos de caso.

Neste algoritmo precisa-se de três listas, uma para representar a "Lista de transições negativas", uma para a lista de atributos de refinamento, e outra para representar a "lista de Ações não executáveis". A *lista de transições negativas* contém as sequências que o verificador assume como inválidas quando o modelo é transformado em propriedade. A *lista de ações não executáveis* contém as sequências de ações para as quais não se conseguiu gerar vetores de testes executáveis na aplicação e, por consequência, são consideradas não executáveis. A *lista de atributos de refinamento* contém os atributos que são utilizados pela ferramenta LTSE para refinar o modelo.

Uma das pré-condições estabelecidas para a execução do algoritmo é que se deve fornecer a especificação do programa**.** Neste trabalho, utilizou-se a lógica FLTL como forma de definição das propriedades. O responsável pela criação das propriedades deve ter conhecimento das ações existentes na aplicação ou que se deseja que a aplicação execute.

No passo denotado pelo símbolo "1" no fluxograma, faz-se a instrumentação do código. Esta instrumentação tem por objetivo fazer uma substituição no código para conseguir-se gerar traços de execução e coletar informações sobre os contextos. É utilizado o mesmo procedimento definido por Duarte (2007). Este procedimento é feito de forma automática e se utiliza de uma gramática de substituição que faz transformações do código em funções de sintaxe equivalente, porém adicionando instruções para a geração dos traços e das informações de contexto.

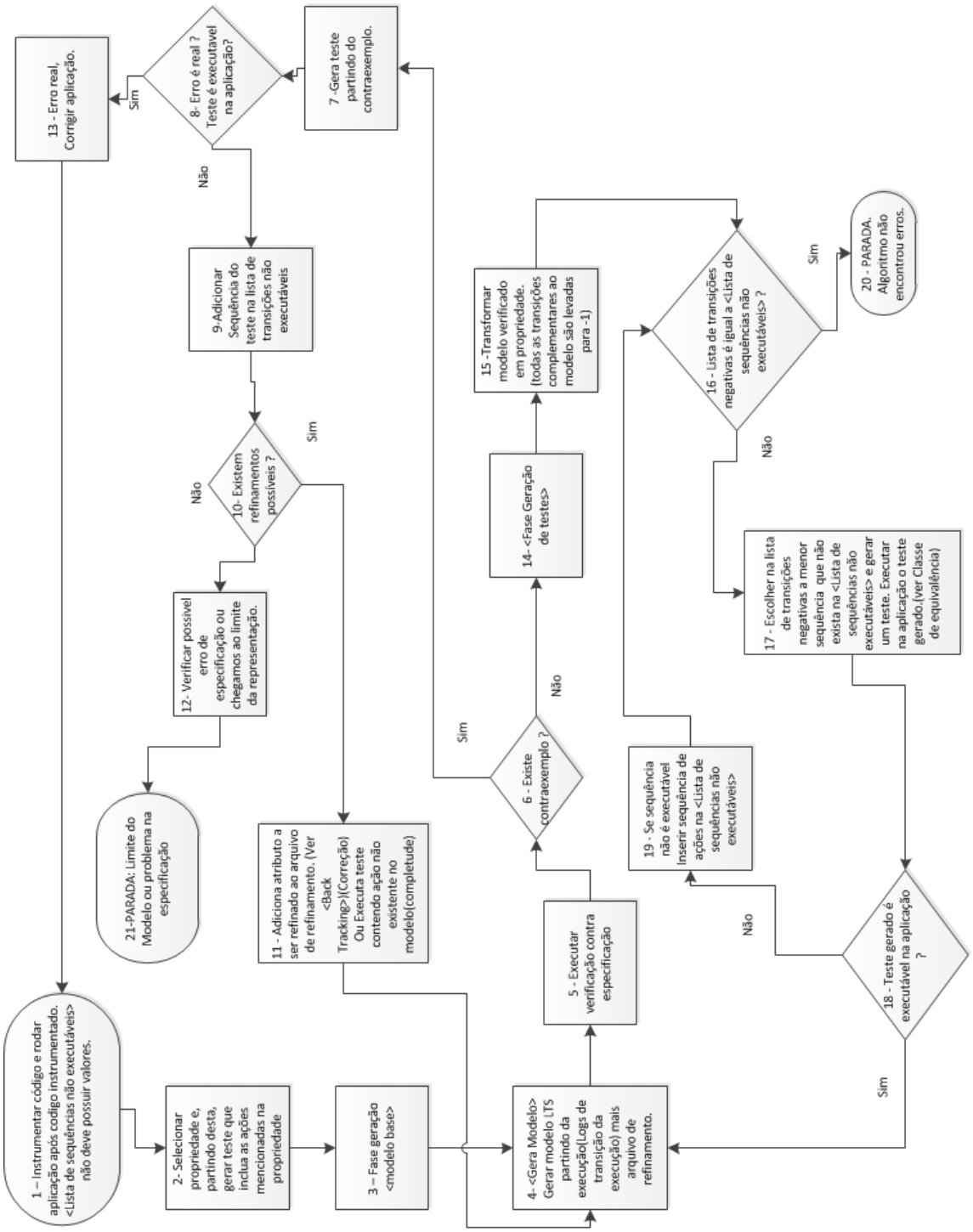

Figura 3.2.1: Fluxograma que descreve o algoritmo proposto

Na figura 3.2.2 podemos ver um exemplo de definição de propriedades em FLTL.

```
fluent AC_ON = <acOn, acOff> initially 0
fluent DOOR_CLOSED = <doorClosed,doorOpen> initially 1
fluent ROOM_HOT = <roomHot, roomCool> initially 0
fluent SYSTEM_OFF = <finished,envcontroller.nextSignal> initially 0

// If room is hot and door is closed, then turn AC on
assert AC_ACTIVE = [](ROOM_HOT && DOOR_CLOSED && !SYSTEM_OFF) -> X AC_ON)
// If room is cool or door is open, then turn AC off
assert SAVE_ENERGY = [](!ROOM_HOT || !DOOR_CLOSED) -> X !AC_ON)
// If system is exiting, AC must be turned off
assert AC_OFF = [](SYSTEM_OFF && AC_ON) -> X !AC_ON)

assert CORRECT_AC = (AC_ACTIVE && SAVE_ENERGY && AC_OFF)
```

Figura 3.2.2: Exemplo de propriedades em FLTL que representa especificação de controlador de ar-condicionado

A instrumentação é feita através da ferramenta *TXL* como discutido por DUARTE (2007). Esta ferramenta faz a instrumentação de um código em linguagem JAVA de forma automática. Após a instrumentação no passo “2”, gera-se um teste inicial de forma que inclua pelo menos uma ação existente nas propriedades a serem validadas. Com o teste gerado, executa-se este teste na aplicação, gerando-se um traço de execução da aplicação. A instrumentação do código gera as informações de contexto e traços na saída de erro padrão utilizada pelo JAVA. Na Figura 3.2.3, podemos ver um exemplo de log de execução após a instrumentação do código.

```
REP_ENTER:(! finished)#true#AirConditioner=9634993#{room_hot=false^door_closed=true^ac_on=false^}#11;
CALL_ENTER:nextSignal#AirConditioner=9634993#EnvController@190d11#{room_hot=false^door_closed=true^ac_on=false^}#0;
MET_ENTER:nextSignal#EnvController=1641745#{in=java.io.BufferedReader@a90653^}#1;
MET_END:nextSignal#EnvController=1641745#1;
CALL_END:nextSignal#AirConditioner=9634993#EnvController@190d11#0;
SEL_ENTER:(message)#ROOM_HOT#AirConditioner=9634993#{room_hot=false^door_closed=true^ac_on=false^}#10;
SEL_ENTER:(! room_hot)#true#AirConditioner=9634993#{room_hot=false^door_closed=true^ac_on=false^}#1;
ACTION:roomHot#AirConditioner=9634993;
SEL_ENTER:(! ac_on && door_closed)#true#AirConditioner=9634993#{room_hot=true^door_closed=true^ac_on=false^}#2;
ACTION:acOn#AirConditioner=9634993;
SEL_END:(! ac_on && door_closed)#AirConditioner=9634993#2;
SEL_END:(! room_hot)#AirConditioner=9634993#1;
SEL_END:(message)#AirConditioner=9634993#10;
REP_END:(! finished)#AirConditioner=9634993#11;
REP_ENTER:(! finished)#true#AirConditioner=9634993#{room_hot=true^door_closed=true^ac_on=true^}#11;
CALL_ENTER:nextSignal#AirConditioner=9634993#EnvController@190d11#{room_hot=true^door_closed=true^ac_on=true^}#0;
MET_ENTER:nextSignal#EnvController=1641745#{in=java.io.BufferedReader@a90653^}#1;
MET_END:nextSignal#EnvController=1641745#1;
CALL_END:nextSignal#AirConditioner=9634993#EnvController@190d11#0;
SEL_ENTER:(message)#ROOM_COOL#AirConditioner=9634993#{room_hot=true^door_closed=true^ac_on=true^}#10;
SEL_ENTER:(room_hot)#true#AirConditioner=9634993#{room_hot=true^door_closed=true^ac_on=true^}#3;
ACTION:roomCool#AirConditioner=9634993;
SEL_ENTER:(ac_on)#true#AirConditioner=9634993#{room_hot=false^door_closed=true^ac_on=true^}#4;
ACTION:acOff#AirConditioner=9634993;
```

Figura 3.2.3: Exemplo de log de execução contendo contexto e ações gerado após execução do código instrumentado.

A “Fase de geração do modelo base”, como descrito no passo “3”, é apenas um identificador que representa o começo da geração do modelo de comportamento que representa a aplicação.

Na fase “4” do algoritmo, utilizam-se os logs de execução existentes até o momento e a lista de atributos de refinamento para gerar um modelo do tipo LTS. Para gerar o modelo se utiliza também a ferramenta LTSE (DUARTE, 2007). Na figura 3.2.4, pode-se ver um exemplo de definição LTS de um modelo que representa uma aplicação para ar-condicionado e que contem as ações *“doorOpen”, “doorClosed”, “roomHot”, “roomCool”, “acOff”, “acOn”, “finished” e “envcontrollert.nextSignal”* e que representam respectivamente os comportamentos: abrir a porta; fechar a porta; sala quente; sala fria; ar-condicionado ligado; ar-condicionado desligado; programa terminado; e o sistema esperando nova entrada.

```
DEFAULT = Q0,
    Q0  = (envcontroller.nextSignal -> Q1),
    Q1  = (doorOpen -> Q2
          |finished -> Q6
          |roomHot -> Q7),
    Q2  = (envcontroller.nextSignal -> Q3),
    Q3  = (doorClosed -> Q0
          |roomHot -> Q4
          |finished -> Q6),
    Q4  = (envcontroller.nextSignal -> Q5),
    Q5  = (finished -> Q6),
    Q6  = (_exit->Q6),
    Q7  = (acOn -> Q8),
    Q8  = (envcontroller.nextSignal -> Q9),
    Q9  = (roomCool -> Q10
          |doorOpen -> Q11
          |finished -> Q12),
    Q10 = (acOff -> Q0),
    Q11 = (acOff -> Q4),
    Q12 = (acOff -> Q6).
```

Figura 3.2.4: Descrição do modelo LTS para um programa “ar-condicionado”

Durante a fase denotada por “5” no fluxograma, é feita a verificação do modelo LTS contra as propriedades definidas. Esta verificação é feita através do uso da ferramenta *Labelled Transition System Analyzer* (LTSA) (MAGEE et al.,2006). Na Figura 3.2.5 podemos ver um exemplo de violação de propriedade em um modelo LTS na representação da ferramenta. O contraexemplo, neste caso, ocorre quando se tem a sequência *“<open,open>”* no modelo, o que leva a uma violação das propriedades. Também se tem contraexemplo quando a sequência *“<exit,close>”* ocorre no modelo.

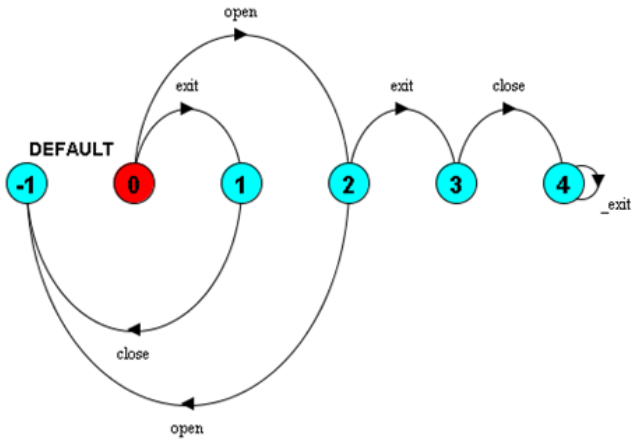

Figura 3.2.5: Violação de propriedades após verificação de modelo LTS na ferramenta LTSA e visualização gráfica do modelo.

No passo “6”, verifica-se se a verificação do modelo gerou algum contraexemplo. Este passo busca garantir a correção e a completude do modelo em relação à especificação e o programa sendo testado e verificado. Caso a verificação do modelo gere um contraexemplo, deve-se verificar se este contraexemplo é espúrio ou real.

Em “7”, gera-se um novo teste partindo-se do contraexemplo encontrado. Este teste é gerado de forma similar à proposta por BEYER et al. (2004). A diferença nesta abordagem é que, como o modelo em questão representa apenas comportamentos da aplicação ou ações, deve-se gerar um vetor de testes a fim de executar a sequência de ações que levam até o estado de erro. Deve-se encontrar uma sequência de entradas na aplicação, através do uso de classes de equivalência, que consiga gerar a mesma sequência de ações do contraexemplo.

Caso se consiga executar este vetor de teste na aplicação, conseguiu-se confirmar a existência do contraexemplo na aplicação, o que implica a identificação de um erro real na aplicação. Esta possibilidade está representada no fluxograma pela notação “13”. Quando um erro real é identificado, é necessária a correção da aplicação e retorna-se ao passo “1” do algoritmo, pois os logs de execução e os testes gerados podem ter sido alterados pela modificação do código.

Caso não se consiga executar ou encontrar nenhum vetor de teste que replique a violação da aplicação, tem-se que o erro não é real. Neste caso, podemos adicionar a sequência de ações à “lista de transições não executáveis”, representado no fluxograma com a notação “9”. Este erro pode ser considerado espúrio, visto que o modelo não está correto, incluindo comportamentos não permitidos na aplicação real. Por outro lado, pode-se também encontrar erros espúrios quando a aplicação ainda não possui informações suficientes sobre determinada ação ou comportamento. Chama-se este tipo de violação de erro de completude, pois a violação ocorre somente no modelo, por este não possuir informação real suficiente relativa à aplicação para satisfazer a propriedade. Em ambos os casos deve-se identificar refinamentos a se fazer no modelo, representado pela notação “10” no fluxograma.

No caso de refinamento para casos de correção, deve-se buscar na aplicação um atributo que consiga adicionar informação no modelo suficiente para eliminar o comportamento inválido. A heurística utilizada neste trabalho é como segue: faz-se

busca reversa nos estados visitados a fim de identificar atributos para utilizar no refinamento. Estes atributos usualmente são utilizados para controlar decisões na aplicação. É importante notar que o atributo de refinamento deve possuir um intervalo de valores discretos, caso contrário, não é garantida a parada do algoritmo, pois o modelo pode crescer indefinidamente. Após se identificar o atributo a ser adicionado, inclui-se o atributo na lista de atributos de refinamento e volta-se para o passo denotado por "4" no fluxograma.

No caso de refinamento para casos de completude, pode-se identificar a existência de erro espúrio de completude através da seguinte heurística: quando a ação que leva ao contraexemplo está sendo adicionada a todos os estados do modelo. Isto quer dizer que não existe informação suficiente no modelo para garantir que o alfabeto das propriedades seja igual ao alfabeto do modelo, assim a ferramenta LTSA assume que se pode executar a ação de qualquer estado existente no modelo LTS. Deve-se gerar um teste adicional contendo a ação em questão. Para gerar este teste, devem-se executar as sequências de ações que surgiram na composição do Modelo contra as Propriedades durante a verificação do modelo. Quando é encontrada a primeira sequência de execução válida contendo a ação que está gerando o contraexemplo, adiciona-se esta sequência aos testes válidos e se retorna ao passo denotado por "4" no fluxograma.

Estes casos de refinamento do modelo são denotados no fluxograma pelo passo "11". Completando-se o refinamento deve-se voltar ao passo "4" do fluxograma e gera-se novamente o modelo com as informações de refinamento adicionadas.

Caso não se consiga identificar um refinamento ou todos os refinamentos possíveis já foram tentados, deve-se parar o algoritmo e verificar uma possível limitação na representação de modelo e propriedade sendo utilizada. Este passo é representado por "21" no fluxograma proposto. Um exemplo para este caso é como segue: o modelo LTS não possui representação de contagem, assim, caso um controle da execução dependa de um certo número de repetições e não se possua um atributo para garantir este controle de repetições, a modelagem não será capaz de representar esta restrição. Se existir uma propriedade que exija certo número de execuções da ação em questão, não será possível representar esta restrição para o modelo LTS.

Este ciclo de geração do modelo se repete até não existirem mais contraexemplos no modelo. Busca-se isto para garantir que os novos testes gerados sejam gerados partindo-se de um modelo correto e completo em relação à especificação, levando-se em conta o nível de abstração deste modelo.

A fase de geração de novos testes é denotada por "14" no fluxograma. O Primeiro procedimento desta fase é transformar o modelo atual em propriedade. Quando se faz este procedimento, a ferramenta LTSA passa a representar todas as transições complementares como contraexemplos. Este procedimento é denotado por "15" no fluxograma. Pode-se ver um exemplo de modelo transformado em propriedade na figura 3.2.6 que segue.

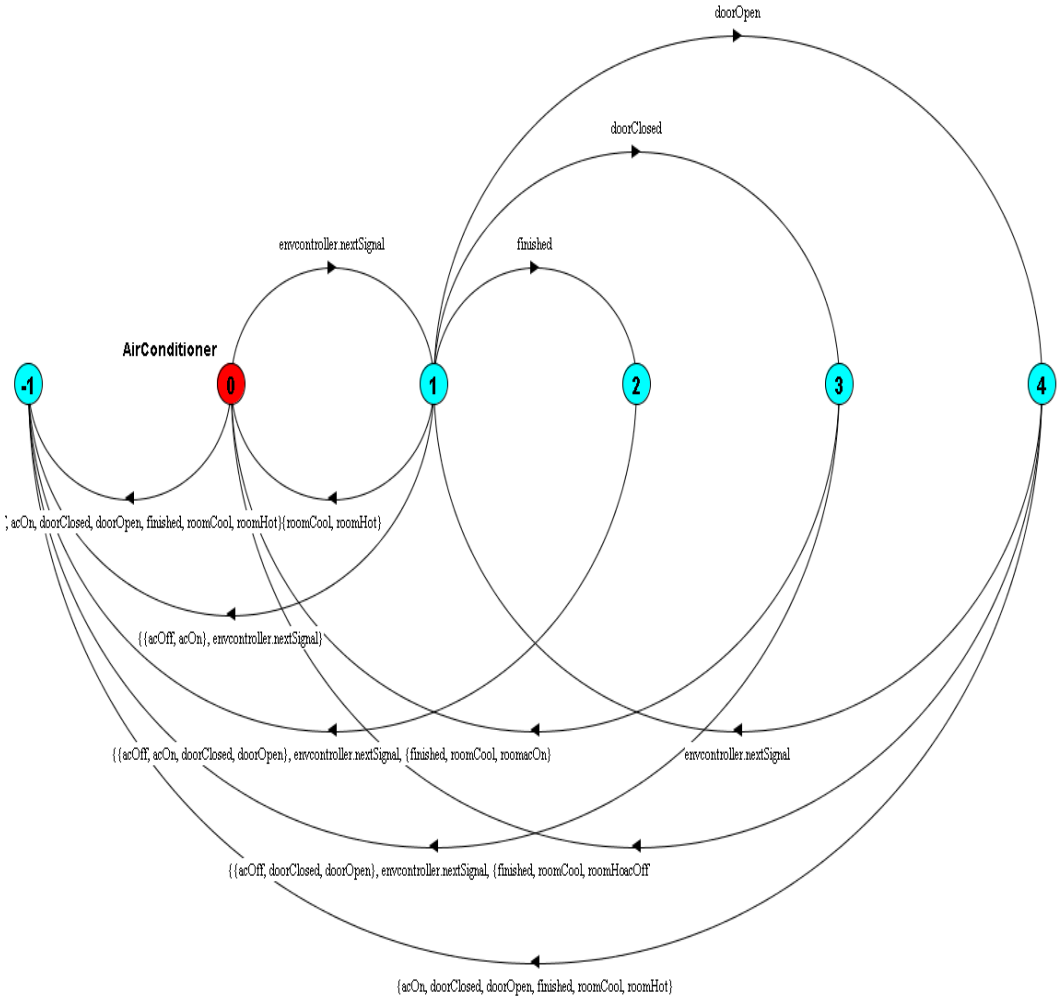

Figura 3.2.6: Exemplo de modelo LTS de um ar-condicionado transformado em propriedade.

Estas transições negativas geradas são adicionadas a uma *lista de transições negativas*. Verifica-se, então, se a lista de transições negativas é igual à lista de ações não executáveis. Esta verificação é denotada por "16" no fluxograma. Ao validar se as listas são iguais, o objetivo é garantir que todos os comportamentos sobre os quais não se possui informação no modelo sejam testados, ou seja, as transições negativas na "lista de transições negativas" que não estão presentes ainda na "lista de ações não executáveis" são comportamentos que podem ou não existir na aplicação, pois ainda não foram observadas.

Caso a *"lista de transições negativas"* possua mais elementos que a *"lista de transições não executáveis"*, então gera-se um teste para a menor sequência de ações existente na *"lista de transições negativas"* e que ainda não exista "lista de transições não executáveis. A geração deste teste é feita da mesma forma que a geração de testes baseado em contraexemplo (BEYER et al., 2004). Busca-se um vetor de teste, partindo do conceito de classes de equivalência, que consiga executar a sequência completa de ações até chegar-se a transição negativa. Este processo está denotado por "17" no fluxograma. Partindo-se da menor sequência existente na lista de transições negativas e que não exista ainda na lista de sequências não executáveis, gera-se o vetor de testes e

continua-se no passo “18”. Se for possível executar na aplicação algum vetor que gere a sequência para chegar-se a uma transição negativa, deve-se adicionar aquele vetor de testes ao conjunto de testes e gerar um *log* de rastro de execução para este vetor. Esta verificação é denotada no fluxograma por “18”. Com o log desta execução adicionado ao conjunto de rastros de execução, deve-se retornar ao passo “4” e gerar um modelo novo com os logs antigos, o log novo e o arquivo de refinamento corrente. Segue a execução do passo “4” seguindo o fluxograma.

Caso não seja possível achar algum vetor de testes que consiga gerar a sequência contida até uma transição negativa sendo testada, deve-se adicionar essa sequência à “lista de ações não executáveis” e voltar para o passo “16”. Este processo está representado por “19” no fluxograma.

O passo “16” é repetido até que a “lista de transições negativas” seja igual à “lista de transições não executáveis”. Assim, pode-se dizer que foi feita uma geração de testes de forma exaustiva partindo-se de informações do modelo.

Quando a “lista de transições negativas” e a “lista de ações não executáveis” forem iguais, pode-se parar o algoritmo. Neste caso, o resultado é que o algoritmo não consegue mais achar inconformidades ou violações no modelo em relação à especificação. Denota-se este ponto de parada por “20” no fluxograma do algoritmo.

4 RESULTADOS EXPERIMENTAIS

Utilizaram-se códigos de programas gerados por terceiros para testar o algoritmo proposto. Primeiramente, se utilizou o código de um editor de texto simples que tinha as ações básicas para controlar a abertura, edição, impressão e salvamento de arquivos. O código e as propriedades definidas para este programa estão descritas no Anexo A. Esta primeira execução foi feita buscando definir o algoritmo.

Em um segundo momento, usou-se o código de um programa de controle de um sistema de ar-condicionado automático. Este sistema deve ser capaz de identificar as condições ambientais de uma sala e, partindo desses estados, tomar ações. As propriedades e o código deste programa podem ser encontrados no Anexo B. Nesta fase, buscou-se a identificação de um código que possuísse erros reais, assim tentou-se evidenciar que o algoritmo conseguiria identificar as falhas e que, depois de removidas, seria possível mapear esta atualização nos modelos gerados posteriormente.

O terceiro estudo de caso utilizou-se de um código sobre um programa *MP3 player*. Este programa deve ser capaz de, partindo de um conjunto de arquivos em um diretório qualquer, gerar uma lista de todos os arquivos e para os arquivos do tipo *MP3* exibir as informações relativas aos atributos da *MP3*, como nome do artista nome do álbum e etc. O código do programa bem como as propriedades definidas podem ser encontradas no anexo C.

Em todos os experimentos, buscou-se identificar os seguintes itens da execução do algoritmo: conjunto de testes iniciais, conjunto de testes gerados, conjunto de ações não executáveis na aplicação, modelo inicial da aplicação, resultado da verificação do modelo contra as propriedades e possivelmente um conjunto de erros identificados e corrigidos.

4.1 Editor

O editor de texto utilizado para este estudo de caso tem como comportamentos identificados os seguintes: tem a capacidade de gerenciar a abertura de um arquivo, pode-se executar a ação de impressão quando um arquivo está aberto, pode-se editar um arquivo aberto, pode-se salvar um arquivo que foi editado e podemos terminar a aplicação, o que deve fechar o arquivo aberto.

As seguintes propriedades que podemos ver na figura 4.1.1 representam as características descritas usando o formalismo FLTL (GIANNAKOPOULOU et al., 2003):

```
// P1
property OpenAndClose = CLOSED,
CLOSED = (open -> OPEN),
OPEN = (close -> CLOSED).

// P2
property SaveOnlyIfEdited = SAVED,
SAVED = (edit -> EDITED),
EDITED = (edit -> EDITED
         |save -> SAVED).
// P3
fluent Open = <open, close> initially 0
assert NotAllowed = (!Open -> !(edit || print || save))
```

Figura 4.1.1: Definição de propriedades para Editor de texto em FLTL

Estas propriedades definem os comportamentos esperados da aplicação. Estes comportamentos são como segue: a propriedade P1 diz que só um arquivo aberto pode ser fechado e somente um arquivo fechado pode ser aberto; a propriedade P2 diz que para um arquivo possa ser salvo ele precisa ter sido editado e, depois de editado, pode também ser editado novamente; e a propriedade P3 diz que um arquivo fechado não pode permitir edição, impressão e salvamento.

Partindo-se de uma das propriedades, gerou-se um teste inicial pseudoaleatório, visando gerar o modelo inicial para representar a aplicação. Este modelo foi gerado a partir do vetor de testes denotado pelas ações <*open,exit*> ou representado pelas classes de equivalência dos inputs da aplicação <*0,4*>. O modelo inicial gerado pode ser visto na figura 4.1.2 a seguir:

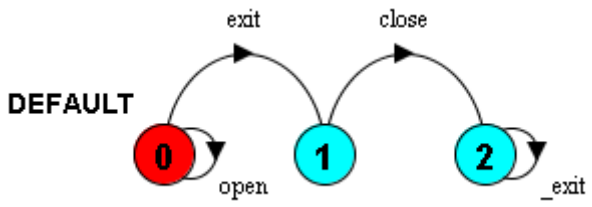

Figura 4.1.2: Modelo LTS para o Editor de Texto

De posse do modelo inicial, verificou-se este contra as propriedades definidas utilizando a ferramenta LTSA (MAGEE et al.,2012) . Foi identificado um erro espúrio, após a geração de testes adicionais, como podemos ver na figura 4.1.3 a seguir:

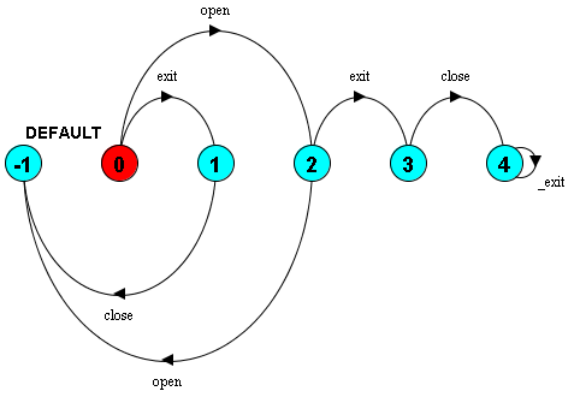

Figura 4.1.3: Erro espúrio identificado após a verificação na sequência <exit,close> e <open,open>

Este erro foi identificado como espúrio, pois ao tentarmos executar a sequência <Open, Open> na aplicação, esta não permite a execução. Assim adicionou-se a sequência <Open, Open> à "*lista de sequências não executáveis*" e buscou-se um atributo para refinamento. Depois de realizado o *backtracking*, identificou-se o atributo "*isOpen*" como atributo para refinarmos o modelo. Partindo do novo atributo de refinamento e os traços de execução existentes, gerou-se o modelo conforme a figura 4.1.4.

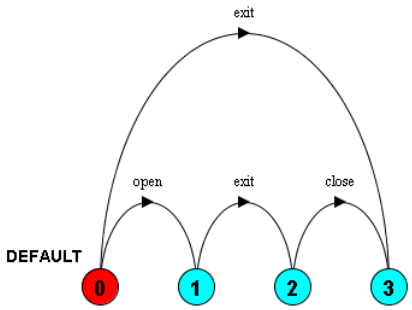

Figura 4.1.4: Modelo do editor refinado utilizando atributo "*isOpen*"

Após a geração do modelo, se verificou novamente as propriedades contra o modelo. Desta verificação foi encontrado um erro espúrio de completude como se pode ver na figura 4.1.5 a seguir:

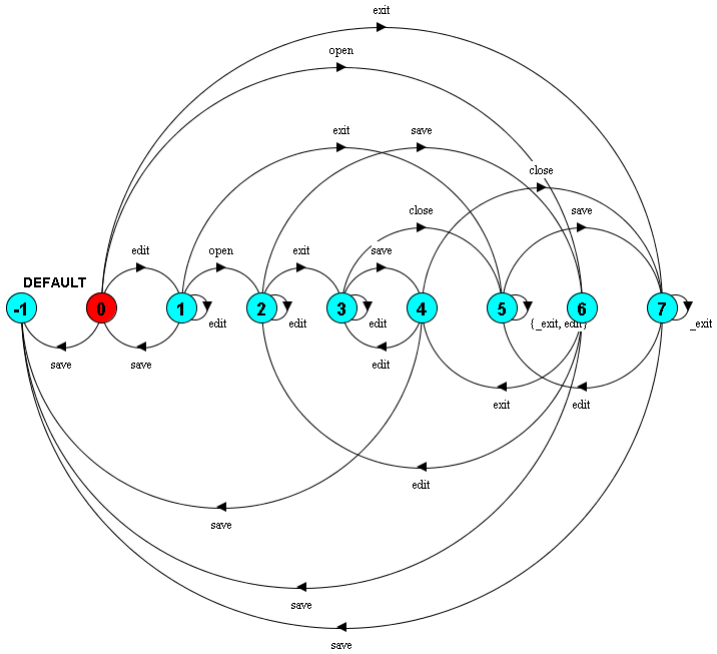

Figura 4.1.5: Erro espúrio de completude na representação da ação *"save"partindo do estado "0"*

Utilizando a heurística para casos em que o erro espúrio é do tipo completude, verificou-se as sequências adicionadas no modelo após a verificação, e com a sequência <*open,edit,save,exit,close*> conseguiu-se gerar a menor sequência que incluía a ação que gerou o contraexemplo. Durante a tentativa de geração de testes, se adicionou à "lista de ações não executáveis" as seguintes sequências: <*save*>, <*edit*>, <*exit,save*>, <*exit,edit*>.

Com a adição deste novo teste ao conjunto de traços de execução gerou-se um novo modelo para a verificação como se pode ver na figura 4.1.6.

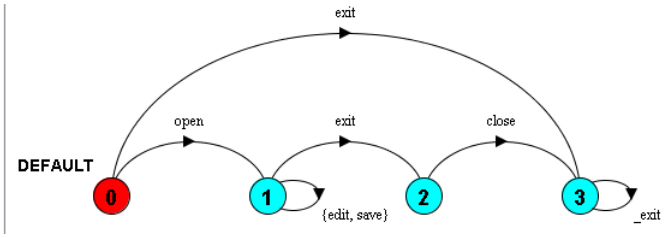

Figura 4.1.6: Adição da ação *"save"* através da heurística de completude

Executou-se novamente a verificação do modelo contra as propriedades após a geração do modelo partindo do novo rastro de execução do teste adicionado. Ainda se pôde identificar um contraexemplo. Pode-se ver o contraexemplo na figura 4.1.7 a seguir:

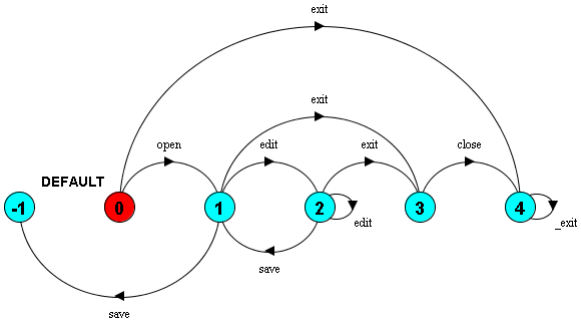

Figura 4.1.7: Contraexemplo identificado da verificação do modelo do editor após adição da ação *"save"*na sequência <*open,save*>

Partindo do contraexemplo identificado tentou-se gerar um contraexemplo partindo da transição negativa <*open,save*>. Esta sequência não foi reproduzida na aplicação real e adicionalmente se incluiu a lista de ações não executáveis a sequência <*open,save*>, assim pode-se considerar como um erro espúrio do tipo "correção". Fazendo o *backtracking* nos estados visitados identificou-se que o atributo para o refinamento é "*isSaved*".. Adicionou-se este atributo e voltamos ao passo "4" do algoritmo para gerar novo modelo com o novo arquivo de refinamento e o conjunto de "LOGS" antigo. O novo modelo gerado pode ser visto na figura 4.1.8 a seguir:

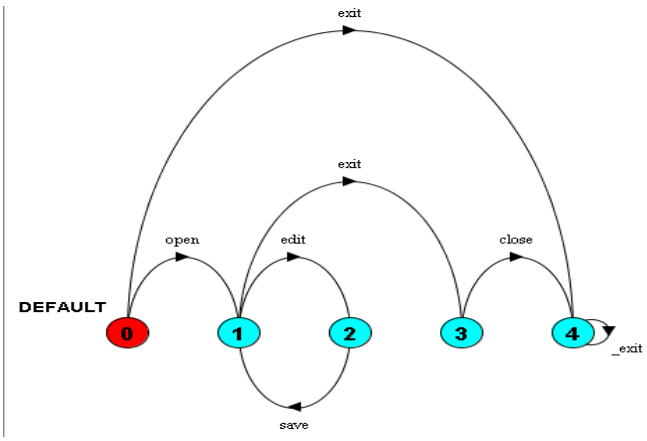

Figura 4.1.8: Modelo LTS do editor atualizado com refinamento "*isSaved*"

Após a geração do novo modelo se executou novamente a verificação das propriedades. Foi identificado desta vez um contraexemplo de completude para a ação "*print*" como se pode ver na figura 4.1.9 a seguir:

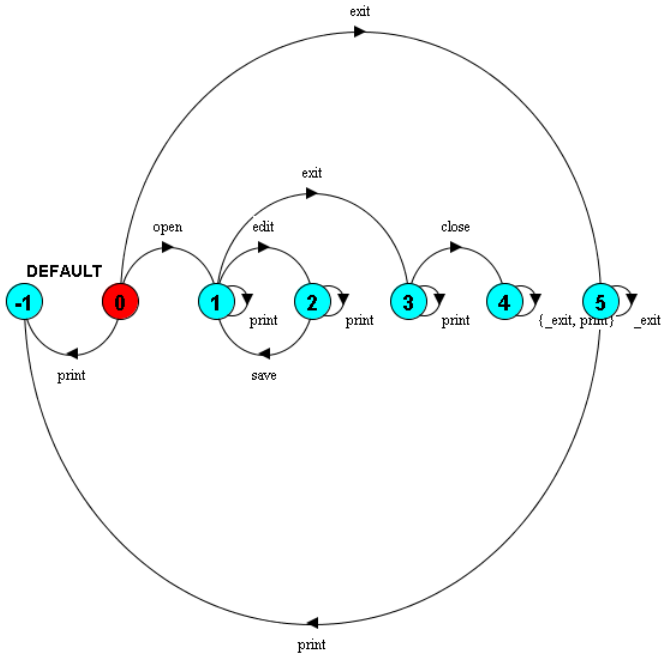

Figura 4.1.9: Contraexemplo espúrio de completude para a ação "*print*"

Utilizando a heurística para se gerar refinamento de completude, verificou-se que a sequência <print> não é executável e pode ser adicionada à lista de transições não executáveis. A primeira sequência válida que incluiu a ação "*print*" é <open,print>. Este teste foi adicionado ao conjunto de testes e voltou-se ao passo "4" para geração de novo modelo com o novo traço de execução que foi adicionado. Pode-se ver o modelo gerado na figura 4.1.10 a seguir:

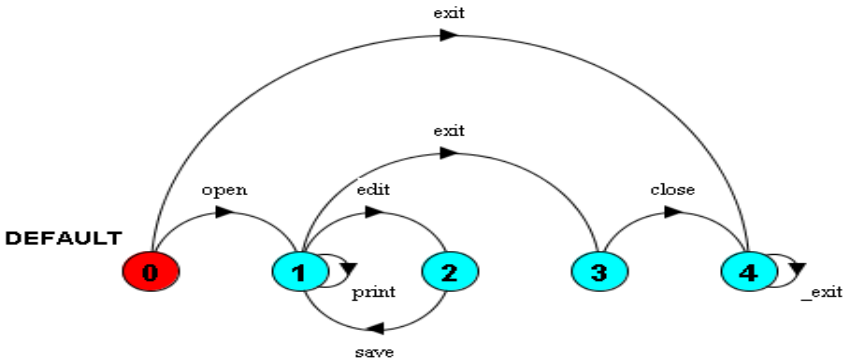

Figura 4.1.10: Modelo gerado com a adição da ação "*print*"

Após a geração do novo modelo se executou a verificação contra as propriedades e nenhum contraexemplo foi gerado. Partiu-se então para o passo "14" do algoritmo, ou seja, a "Fase de Geração de Testes". Partindo do modelo se transformou este em propriedade resultando no modelo representado na figura 4.1.11:

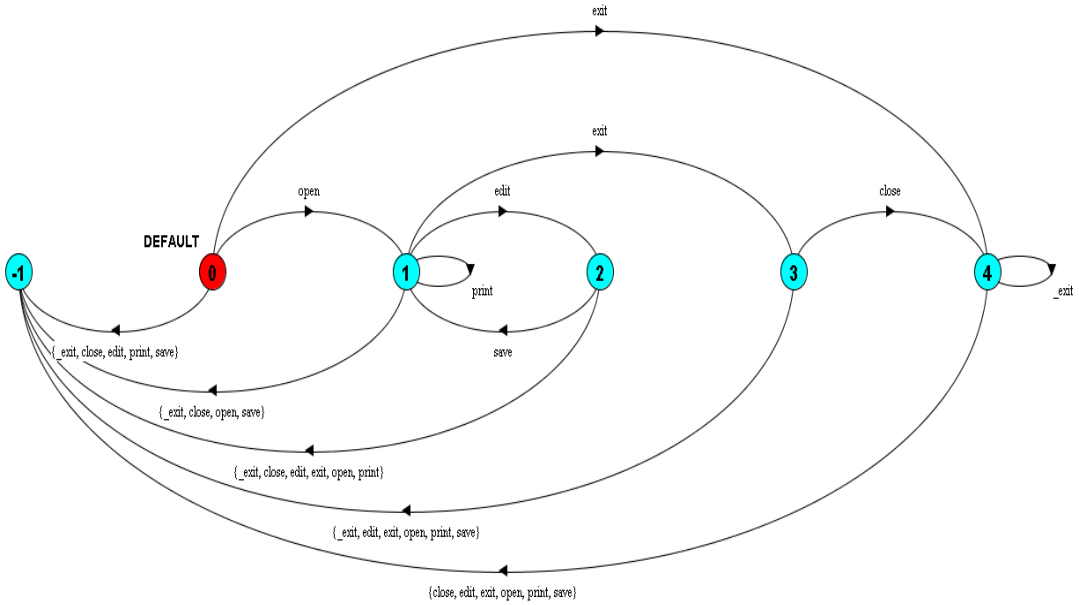

Figura 4.1.11: Modelo LTS do editor de texto transformado em propriedade

Neste ponto, a lista de ações não executáveis não é igual à lista de transições negativas. Assim, tentou-se gerar testes, partindo dos contraexemplos ou transições negativas, para as sequências de tamanho unitário existentes na lista de transições negativas, porém não foi possível gerar vetor de testes que fosse executável na aplicação. Então se adicionou as sequências *<close>, <edit>, <print>, <save>* à lista de sequências não executáveis.

Repetiu-se o procedimento para as sequências existentes na lista de transições negativas de tamanho 2, ou seja, iniciadas por *<open>*. A lista de transições negativas já possuía *<open,open> e <open,save>*. Tentou-se gerar um teste no baseado na transição negativa para <open,close> mas também não foi possível encontrar uma classe de equivalência que pudesse gerar este teste, assim foi adicionada à lista de transições não executáveis. Partindo das sequências iniciadas por *<exit>* de tamanho 2 não foi possível gerar testes partindo dos contraexemplos utilizando classes de equivalência, assim adicionou-se à lista de ações não executáveis as seguintes sequências: *<exit,edit>, <exit,open>, <exit,print>* e *<exit,save>*. A lista de ações não executáveis até este momento continha as seguintes entradas: *(ListaDANE = {<open,open>, <exit,close>, <save>, <edit>, <exit,save> ,<exit,edit> ,<open,save> ,<print> ,<open,close>, <exit,edit>, <exit,open>, <exit,print>* e *<exit,save>}).*

Para sequências de tamanho 3, iniciadas por <open,edit>, foram identificados casos de teste partindo de contraexemplos e usando classes de equivalência para *<open,edit,edit>* e *<open,edit,print>*. Para ambos os casos se adicionou os traços de execução ao conjunto de testes e gerou-se um novo modelo no passo "18" e retornando ao passo "4". Pode-se ver o modelo gerado na figura 4.1.12:

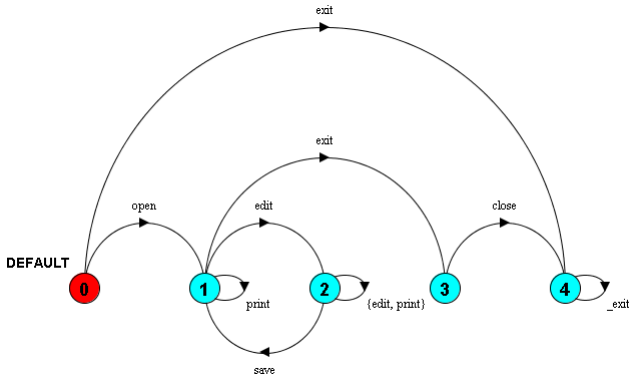

Figura 4.1.12: Modelo gerado após a adição de testes para sequências de tamanho 3

Após a geração de novo modelo este foi verificado contra as propriedades no passo "5" e nenhum contraexemplo foi gerado. Continuou-se então no passo "14", "15" e "16" onde se verificou que a lista de transições negativas ainda não é igual à lista de sequências não executáveis. Desta forma se continuou o passo "17" com as sequências de tamanho 3 ainda não existentes na lista de transições não executáveis. Conseguiu-se gerar teste para a sequência *<open,edit,exit>*, assim verificando novo comportamento na aplicação. Adicionado o traço de execução em "18" e voltamos para o passo "4" para geração de novo modelo. Pode-se verificar o modelo gerado na figura 4.1.13 a seguir:

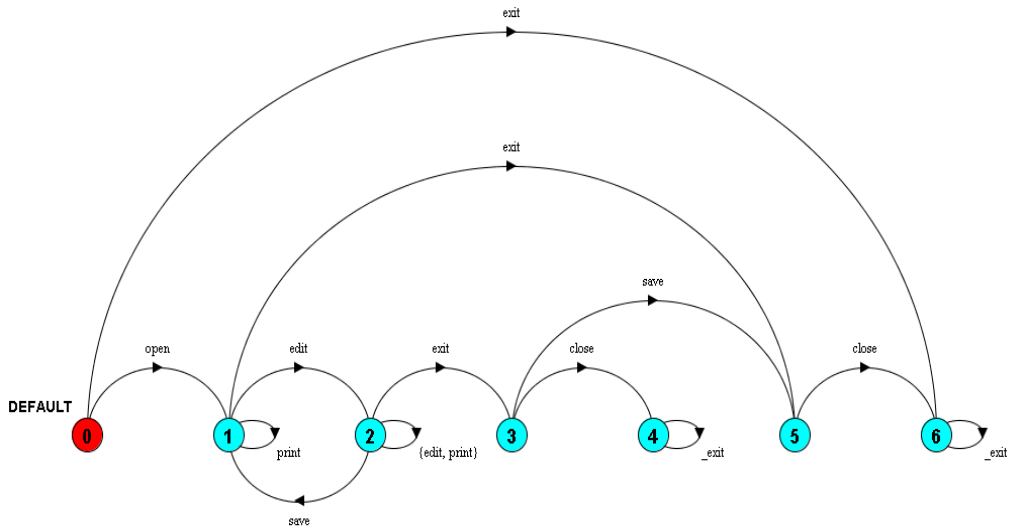

Figura 4.1.13: Adição de novo comportamento identificado após a sequência *<open,edit,exit>*

Após a geração do modelo foi verificado o novo modelo gerado contra as propriedades e nenhum contraexemplo ou violação foi identificado. Continuou-se a execução do algoritmo em "14", "15" e "16". Não se conseguiu gerar testes partindo dos contraexemplos para sequências restantes na lista de transições negativas, assim todas foram adicionadas à lista de sequências não executáveis. A lista final ficou como segue: *(ListaDANE={<open,open>, <exit,close>, <save>, <edit>, <exit,save>, <exit,edit>, <open,save>, <print>, <open,close>, <open,edit,open>,*

<open,edit,close>, <open,edit,exit,edit>, <open,edit,exit,exit>, <open,edit,exit,open>, <open,edit,exit,print>, <open,edit,exit,save,edit>, <open,edit,exit,save,exit>, <open,edit,exit,save,open>, <open,edit,exit,save,print>, <open,edit,exit,save,save>}).

A Lista de testes final gerada ficou como segue: (Lista de testes = { *<open,exit>, <open,print>, <open,edit,edit>, <open,edit,print>, <open,edit,exit>, <open,edit,save,exit,close>*}).

Nesta aplicação de estudo, não se conseguiu identificar falhas de código na aplicação em relação à especificação. Pode-se verificar a adição de vários testes de forma sistemática a um conjunto de testes, visando aumentar a cobertura dos testes sobre a aplicação, bem como um conjunto de ações que foram definidas como não executáveis na aplicação. O modelo LTS final da aplicação pôde ser gerado e verificado contra a especificação e não gerou contraexemplos.

A tabela 4.1.1 a seguir descreve as informações coletadas durante a execução do algoritmo sobre o estudo de caso do editor de textos.

Tabela 4.1.1: Tabela consolidada com os resultados da execução do algoritmo no primeiro programa usado como estudo de caso

Número de testes iniciais	1
Número de testes válidos gerados	6
Número de testes negativos gerado	20
Número de problemas identificados/Número de erros existentes	0/0
Número de ciclos executados do algoritmo	5+16
Número de estados do modelo final	6
Número de transições no modelo final	13

Estes resultados indicam que partindo de apenas um teste inicial se, conseguiu partindo das informações contidas no modelo e o algoritmo proposto, a geração de cinco testes novos. Nenhuma falha foi identificada, porem o código já estava de acordo com a especificação. Como o número de diferentes possibilidades de combinação do programa não é grande o número de ciclos que foi necessário para a sua parada não foi relativamente alto. O número de estados do modelo final, bem como o número de transições, também não foi um número representativamente grande, mas representa o comportamento do programa de forma satisfatória.

4.2 ACC – Controle de Ar-Condicionado

O programa de controle de ar-condicionado, utilizado como segundo estudo de caso, tem as seguintes características: existem quatro comportamentos diferentes de entrada, a saber, "*ROOM_HOT*" para identificar o comportamento quando o ambiente tornou-se quente, "*ROOM_COOL*" para identificar o comportamento quando a sala fica fria,

"*DOOR_OPEN*" para identificar o comportamento a porta ter sido aberta, "*DOOR_CLOSED*" para identificar o comportamento onde a porta ter sido fechada e "*OFF*" que representa o desligamento do sistema.

Com estes comportamentos mapeados definiram-se as propriedades, ou requisitos do sistema, da seguinte forma: o ar-condicionado só poderá estar ligado, representado pelo fluente "*AC_ON*", quando a sala estiver quente e a porta estiver fechada; sempre que a porta estiver aberta ou a sala ficar fria o ar-condicionado deve ser desligado, ou seja, fluente "*AC_ON*" deve ser falso; e quando o sistema for desligado o ar-condicionado deve ser desligado caso este esteja ligado. Definiram-se as propriedades utilizando FLTL (GIANNAKOPOULOU et al., 2003) como se pode verificar na figura 4.2.1 a seguir.

```
fluent AC_ON = <acOn, acOff> initially 0
fluent DOOR_CLOSED = <doorClosed,doorOpen> initially 1
fluent ROOM_HOT = <roomHot, roomCool> initially 0
fluent SYSTEM_OFF = <finished,envcontroller.nextSignal> initially 0

// If room is hot and door is closed, then turn AC on
assert AC_ACTIVE = [](ROOM_HOT && DOOR_CLOSED && !SYSTEM_OFF) -> X AC_ON)
// If room is cool or door is open, then turn AC off
assert SAVE_ENERGY = [](!ROOM_HOT || !DOOR_CLOSED) -> X !AC_ON)
// If system is exiting, AC must be turned off
assert AC_OFF = [](SYSTEM_OFF && AC_ON) -> X !AC_ON)

assert CORRECT_AC = (AC_ACTIVE && SAVE_ENERGY && AC_OFF)
```

Figura 4.2.1: Propriedades em FLTL da aplicação controladora do ar-condicionado

Gerou-se um modelo inicial partindo da propriedade "*AC_ACTIVE*" seguindo os passos um até três do algoritmo. Criou-se um teste com o seguinte vetor de entrada <*room_hot,finished*>. A representação do modelo inicial pode ser definida pelo modelo LTS como se pode ver na figura 4.2.2.

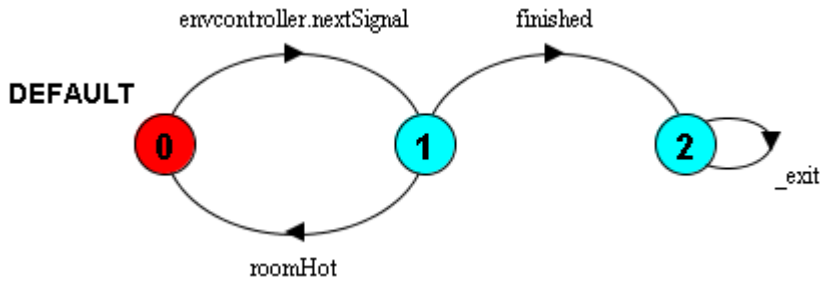

Figura 4.2.2: Modelo LTS inicial representando a aplicação ar-condicionado

Executou-se a verificação contra as propriedades seguindo o passo "5" do algoritmo. Encontrou-se contraexemplo nesta verificação contra a propriedade *"AC_ACTIVE"*, a

sequência <*roomHot,envcontroller.nextSignal*> é considerada o contraexemplo. Pode-se ver o resultado da verificação como na figura 4.2.3 a seguir.

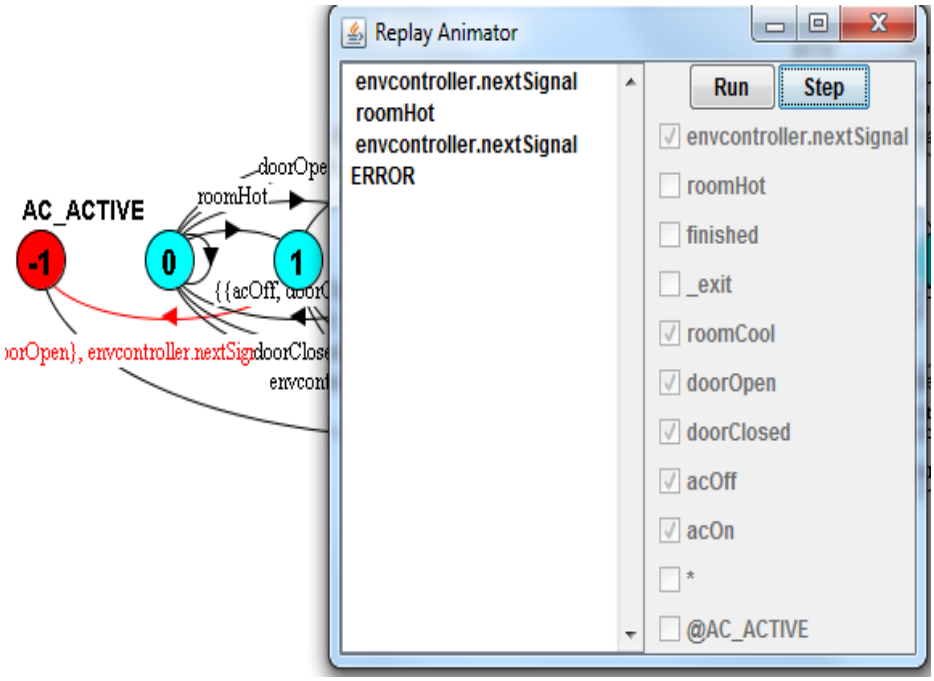

Figura 4.2.3: Contraexemplo ou violação de propriedade ao verificar modelo Inicial

Assim, criou-se um vetor de teste partindo do contraexemplo para verificar a execução da sequência de erro na aplicação, neste caso não seria necessário, pois o teste a ser gerado é trivialmente igual ao teste utilizado anteriormente para criar o modelo inicial. Utilizou-se o vetor de teste <*roomHot*> e, após a entrada deste input, a aplicação permitiu executar a ação <*envcontroller.nextSignal*>, que representa a possibilidade de um usuário alterar através de um input o estado do ambiente. Pela propriedade "*AC_ATIVE*", esperava-se que ao ocorrer o comportamento "*roomHot*", como o valor inicial de "*DOOR_CLOSED*" é verdadeiro, o sistema deveria gerar o comportamento "*acOn*". Neste caso seguimos a sequência "7, 8 e 13" do algoritmo e foi identificado um erro na aplicação que se deve corrigir antes de prosseguir com o algoritmo.

A correção do código do programa buscando a correção do problema pode ser verificada na figura 4.2.4 a seguir.

```
        switch (message) {
      case ROOM_HOT:

                    if (!room_hot) {

                                room_hot = true;
                                #action:"roomHot";
                                System.out.println ("-> Room hot");
                                if (!ac_on && door_closed){
                                        ac_on = true;
                                        #action:"acOn";
                                        System.out.println ("-> AC on");
                                }

                    }
                                break;
```

Figura 4.2.4: Correção no código da aplicação para resolver problema real

Após a correção voltou-se para o passo inicial do algoritmo e gerou-se um novo modelo inicial partindo do mesmo vetor de testes previamente utilizado. No passo "4" gerou-se um novo modelo como se pode verificar na figura 4.2.5 a seguir.

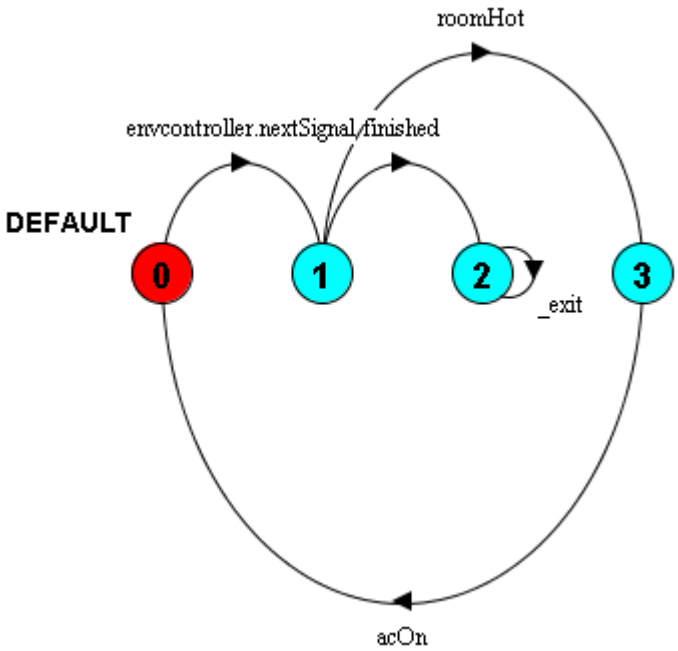

Figura 4.2.5: Novo modelo gerado após correção do código.

Após a verificação do novo modelo contra as propriedades encontrou-se um contraexemplo contra o comportamento "*ROOM_COOL*". A ferramenta identificou o contraexemplo de sequência *<envcontroller.nextSignal,roomHot,roomCool>*. Pelos passos 7 e 8, tentou-se gerar um vetor de testes mas não se conseguiu achar classe de equivalência para atingir a sequência de erro na aplicação. Verificou-se existir um refinamento de completude e adicionou-se um teste que inclua a ação "*roomCool*".

Adicionou-se o rastro de execução e gerou-se um novo modelo como se pode verificar na figura 4.2.6 a seguir.

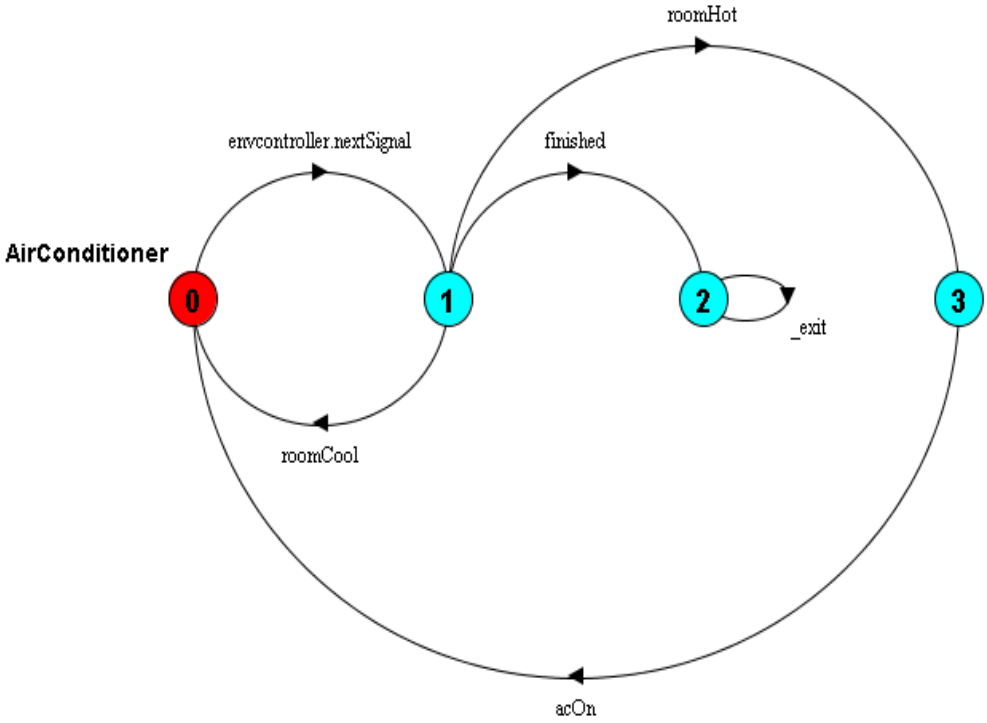

Figura 4.2.6: Modelo adicionando teste para refinamento de completude para "*roomCool*"

Verificou-se novamente o modelo contra as propriedades e foi identificado novo contraexemplo de completude, onde não se possui informação ainda sobre o comportamento "*doorOpen*". O seguinte vetor foi utilizado para refinar o modelo quanto à completude: <doorOpen>. Adicionou-se este teste e foi gerado um novo modelo como se pode ver na figura 4.2.7 a seguir.

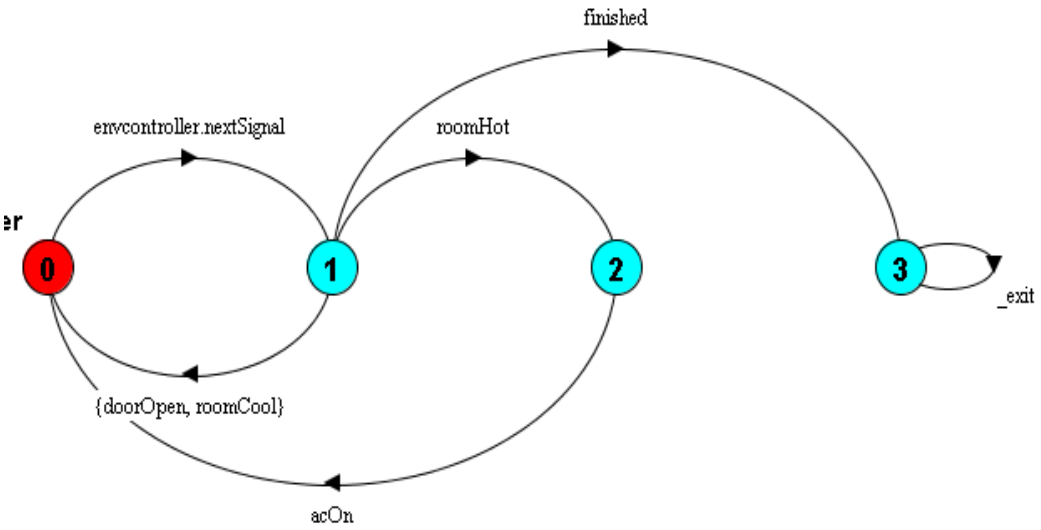

Figura 4.2.7: Adição de comportamento "*doorOpen*"

Voltou-se ao passo “5” e verificou-se novamente o modelo contra as propriedades. Novamente identificou-se um erro espúrio de completude, pois não foi possível executar a sequência de erro na aplicação. A partir das transições do modelo identificou-se o vetor de teste: <*doorOpen,doorClosed*>. Adicionado este novo traço de execução se gerou novamente o modelo e que se pode verificar conforme a figura 4.2.8 a seguir.

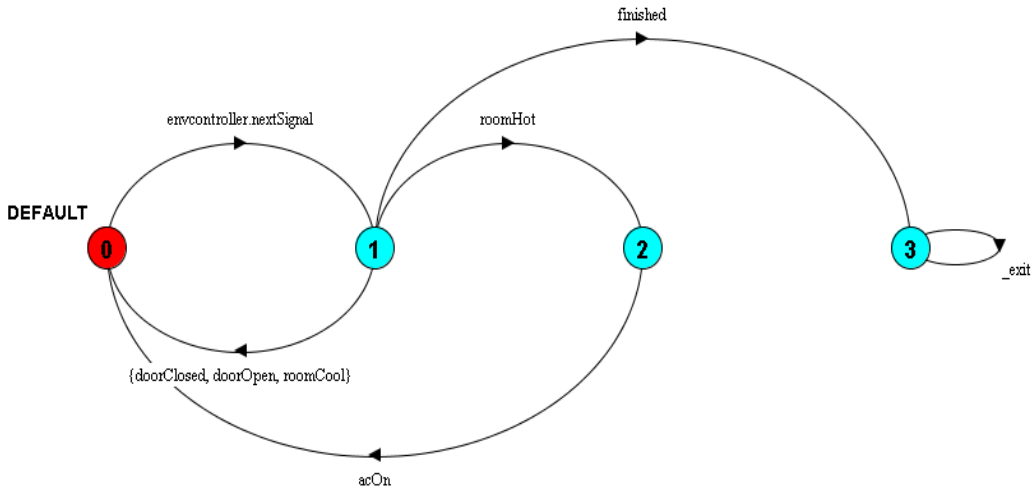

Figura 4.2.8: Adição do comportamento “*doorClosed*” ao modelo após refinamento de completude.

Verificaram-se novamente as propriedades e identificou-se outro erro de completude no modelo, pois a ação “*acOff*” não consta no modelo. Não foi possível criar um vetor de teste e executar a sequência identificada como violação na aplicação. A partir do modelo se conseguiu identificar o seguinte vetor de teste: <*roomHot,acOn,doorOpen*>. Gerou-se um novo modelo conforme a figure 4.2.9 a seguir.

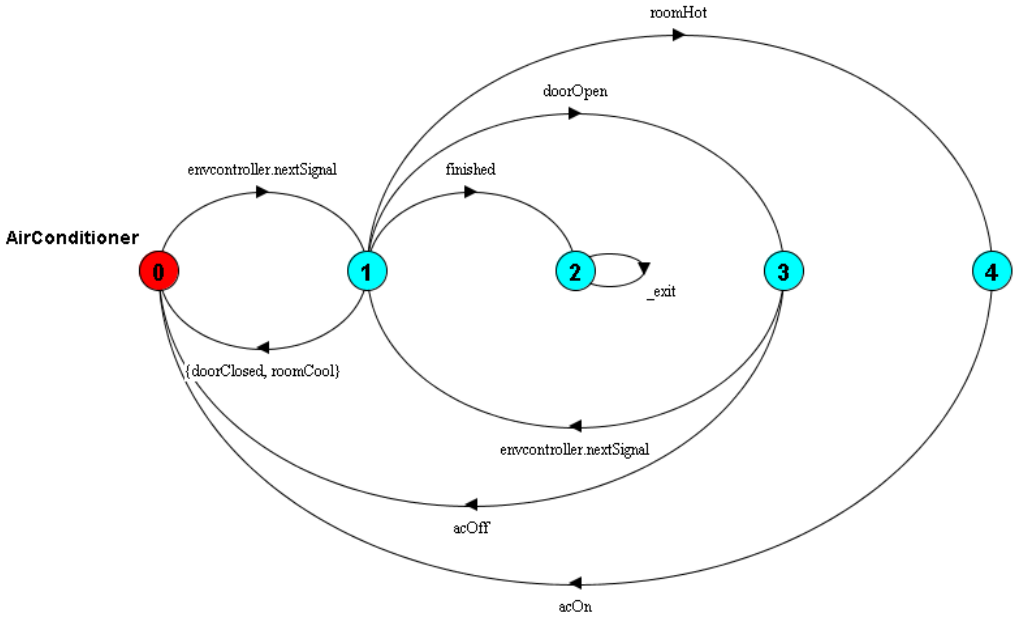

Figura 4.2.9: Adição do comportamento "*acOff*" para refinamento de completude

Verificou-se contra as propriedades o novo modelo gerado e um contraexemplo foi identificado. Este contra exemplo é a sequência *<envcontroller.nextSignal,roomHot, acOn,envcontroller.nextSignal,doorOpen,acOff, envcontroller.nextSignal, doorClosed> envcontroller.nextSignal*>. Não foi possível gerar um vetor de teste que para esta sequência, assim verificou-se que um refinamento de correção é necessário. Através do "*bracktracking*" identificaram-se os atributos "*door_closed*" e "room_hot" como atributos de refinamento. Adicionou-se este atributo e o se gerou novo modelo. Pode-se verificar o modelo como na figura 4.2.10 a seguir.

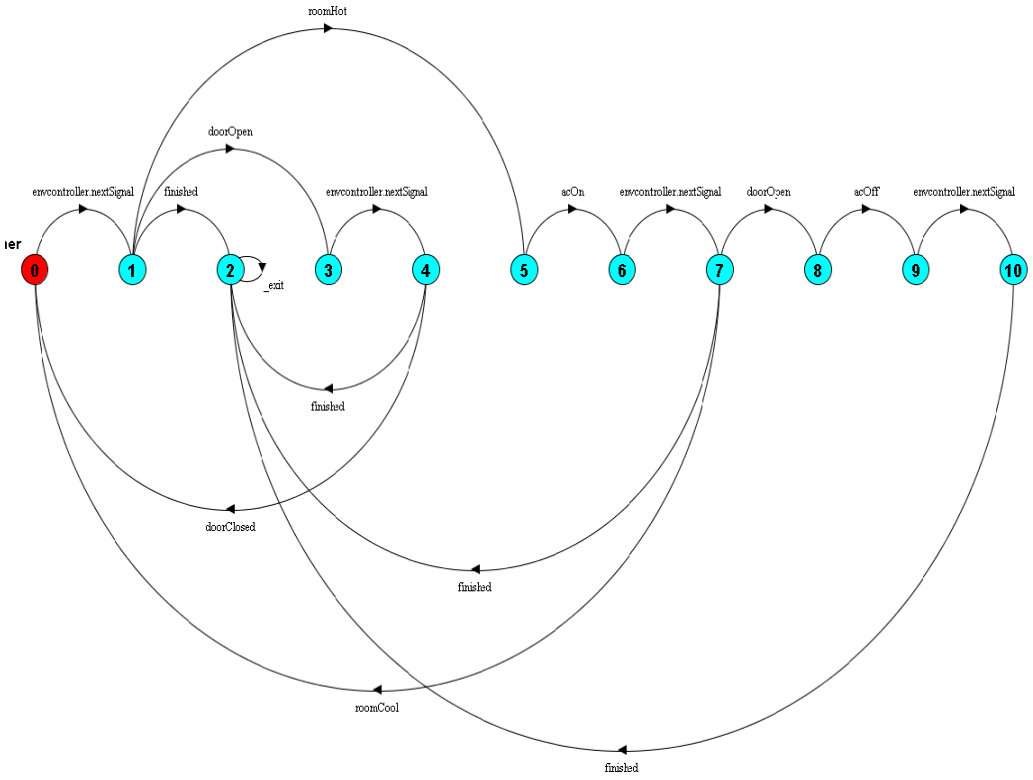

Figura 4.2.10: Adição de atributos de refinamento "*door_closed*"e "*room_hot*"

Verificou-se novamente contra as propriedades o novo modelo e um contraexemplo foi verificado contra a propriedade "*SAVE_ENERGY*". A violação identificada foi a seguinte: <*envcontroller.nextSignal, roomHot, acOn, envcontroller.nextSignal, roomCool envcontroller.nextSignal*>. Gerou-se um vetor de testes da seguinte forma <*roomHot,roomCool*> e foi possível a execução da sequência da violação na aplicação.

Assim identificou-se um erro real na aplicação. A seguir na figura 4.2.11 pode-se verificar a correção do código para este erro real.

```
case ROOM_COOL: if (room_hot) {
                                    room_hot = false;
                                    #action:"roomCool";
                                    System.out.println ("-> Room cool");
                                    if (ac_on){
                                        ac_on = false;
                                        #action:"acOff";
                                        System.out.println ("-> AC off");
                                    }
                                }
                                break;
```

Figura 4.2.11: Correção de código para erro real identificado

Após a correção do código, voltou-se ao inicio do algoritmo e rodou-se o conjunto de testes identificados até aqui. Com o conjunto de traços gerados e o arquivo de

refinamento, gerou-se novamente o modelo inicial. O novo modelo inicial é como se pode verificar na figura 4.2.12 a seguir.

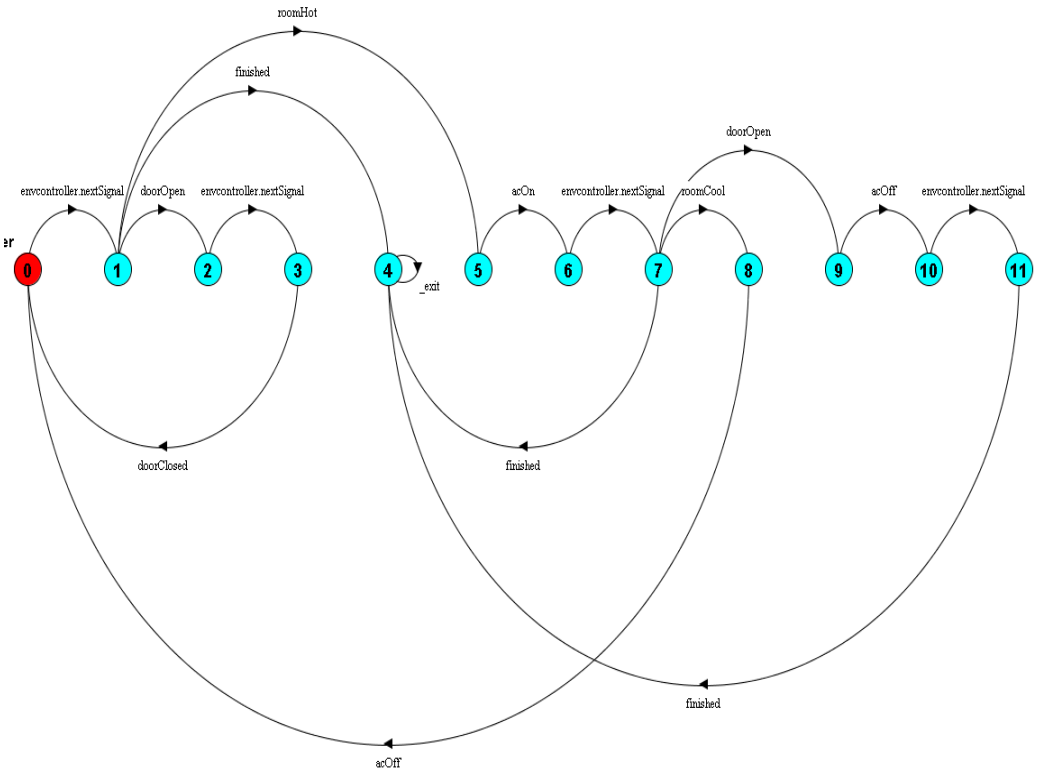

Figura 4.2.12: Modelo gerado após correção de segundo erro real.

Verificaram-se novamente as propriedades contra o novo modelo gerado e uma violação foi identificada contra a propriedade "*AC_OFF*". A sequência de ações identificada é a seguinte: <*envcontroller.nextSignal, roomHot acOn, envcontroller.nextSignal, finished, _exit*>. Tentou-se então gerar um vetor de testes que executasse esta sequência na aplicação. Foi possível gerar através de classes de equivalência e rodar na aplicação o seguinte vetor de testes: <*roomHot,acOn,finished*>. Assim foi identificada mais uma falha real na aplicação onde a correção no código pode ser vista pelo trecho da figura 4.2.13 na figura a seguir.

```
case OFF:   finished = true;
                    #action:"finished";
                    if (finished && ac_on){
                    ac_on = false;
                    #action:"acOff";
                    System.out.println ("-> AC off");
                    }
                    break;
```

Figura 4.2.13: Correção de código para propriedade "*AC_OFF*"

Após a correção do código, voltou-se ao inicio do algoritmo e rodou-se o conjunto de testes identificados até aqui. Com o conjunto de traços gerados e o arquivo de refinamento, gerou-se novamente o modelo inicial. O novo modelo inicial é como se pode verificar na figura 4.2.14 a seguir.

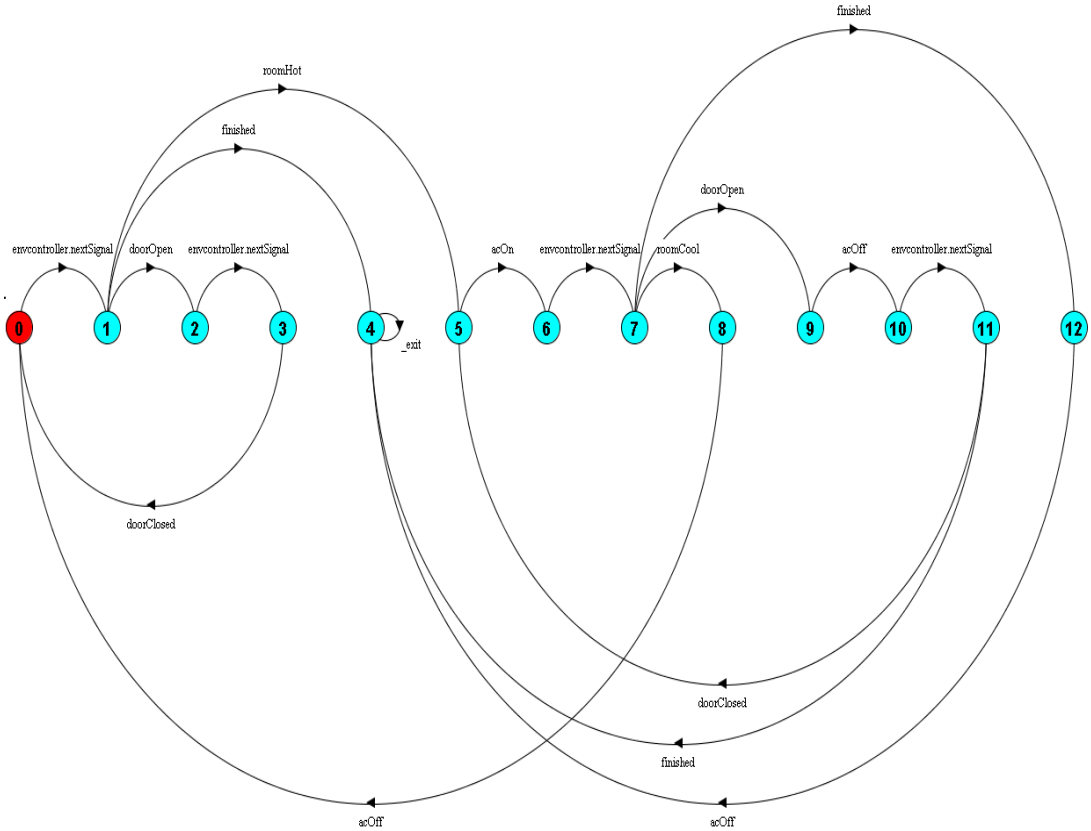

Figura 4.2.14: Modelo gerado após correção de violação de "*AC_OFF*"

Verificaram-se novamente as propriedades e desta vez não se encontrou mais contraexemplos. Assim se pode passar para a fase de "14" do algoritmo. Transformou-se o modelo em propriedade e as "transições negativas" foram adicionadas para comportamentos que ainda não se tem conhecimento. Começando pelas sequências de tamanho unitário, trivialmente não se conseguiu criar vetores de testes e executar as transições na aplicação, pois a primeira ação da aplicação é sempre a espera por um comando denotado por *<envcontroller.nextSignal>*. Assim adicionou-se as sequências *<acOff, acOn, doorClosed, doorOpen, finished, roomCool, roomHot>* a lista de ações não executáveis.

Para as sequências de ações de tamanho "dois", também não se conseguiu gerar vetores de teste e executar na aplicação as sequências. Assim, adicionou-se as sequências *{<envcontroller.nextSignal,<envcontroller.nextSignal>* e *<envcontroller.nextSignal, {_exit, acOff, acOn, doorClosed, roomCool}>}* a lista de ações não executáveis.

Para as sequências de ações de tamanho "três" iniciadas pelo prefixo *<envcontroller.nextSignal,doorOpen>* não se conseguiu gerar vetores de teste ou executar qualquer ação diferente de *envcontroller.nextSignal*. Assim, se pode adicionar

à lista de ações não executáveis as sequências: *<envcontroller.nextSignal,doorOpen, {_exit, acOff, acOn, doorClosed, doorOpen, finished, roomCool, roomHot}>*.

Para as sequências de ações de tamanho "três" iniciadas pelo prefixo *<envcontroller.nextSignal,finished>* não se conseguiu gerar vetores de teste ou executar qualquer sequência de ações. Assim, se pode adicionar à lista de ações não executáveis as sequências: *<envcontroller.nextSignal,finished,{acOff, acOn, doorClosed, doorOpen, envcontroller.nextSignal,finished, roomCool, roomHot}>*.

Para as sequências de ações de tamanho "três" iniciadas pelo prefixo *<envcontroller.nextSignal,roomHot>* não se conseguiu gerar vetores de teste ou executar qualquer sequência de ações que não "*acOn*". Assim, se pode adicionar a lista de ações não executáveis as seguintes sequências: *<envcontroller.nextSignal,roomHot, {_exit, acOff, doorClosed, doorOpen, envcontroller.nextSignal, finished, roomCool, roomHot}>*.

Para as sequências de ações de tamanho "quartro" iniciadas por *<envcontroller.nextSignal,doorOpen, envcontroller.nextSignal>*, se conseguiu gerar vetores de teste para *roomHot* e *finished.* Assim adicionaram-se os testes ao conjunto de traços e um novo modelo foi gerado. Pode-se verificar o novo modelo gerado na figura 4.2.15 a seguir:

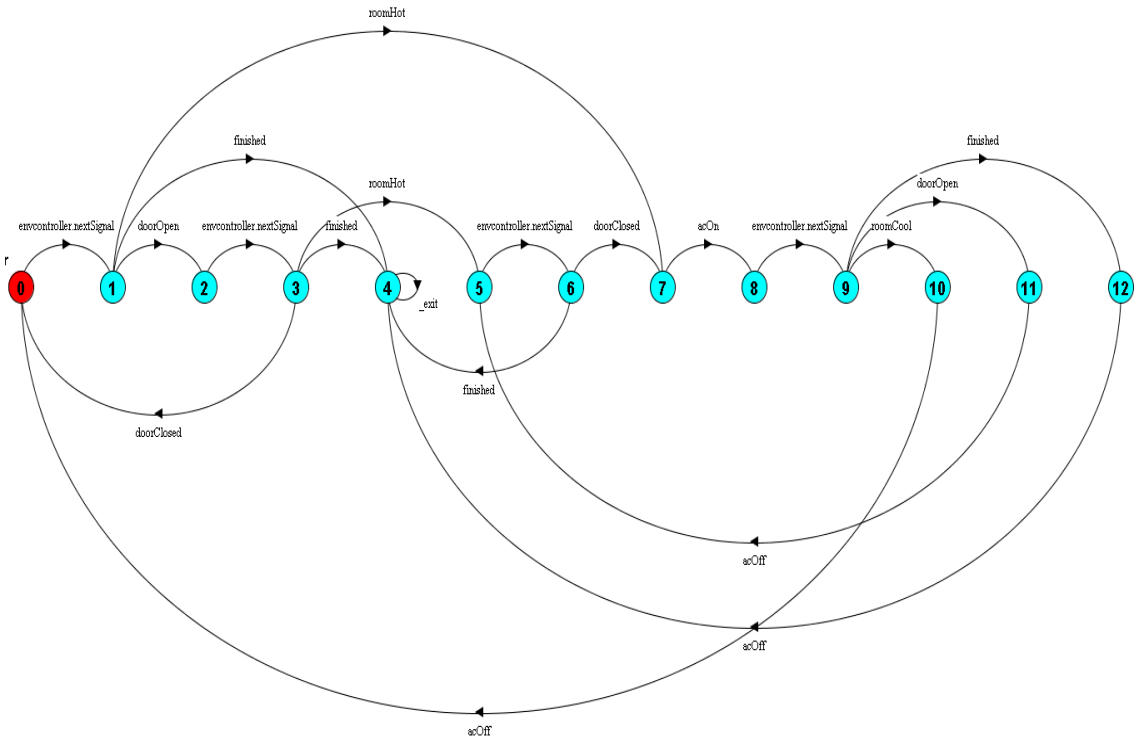

Figura 4.2.15: Adição de comportamentos partindo de testes gerados pelo contraexemplo

Validaram-se as propriedades e não foi identificada nenhuma violação após a adição do comportamento. Repetiu-se o procedimento para as sequências de todos os tamanhos presentes na lista de transições negativas. Somente adicionou-se mais um sequência de comportamentos através do vetor de teste *<doorOpen, roomHot, roomCool>*. A

descrição do modelo LTS final pode ser vista da forma descrita na figura 4.2.16 a seguir.

```
AirConditioner = Q0,
    Q0  = (envcontroller.nextSignal -> Q1),
    Q1  = (doorOpen -> Q2
          |roomHot -> Q3
          |finished -> Q4),
    Q2  = (envcontroller.nextSignal -> Q5),
    Q3  = (acOn -> Q6),
    Q4  = (_exit->Q4),
    Q5  = (doorClosed -> Q0
          |finished -> Q4
          |roomHot -> Q7),
    Q6  = (envcontroller.nextSignal -> Q8),
    Q7  = (envcontroller.nextSignal -> Q9),
    Q8  = (doorOpen -> Q10
          |roomCool -> Q11
          |finished -> Q12),
    Q9  = (roomCool -> Q2
          |doorClosed -> Q3
          |finished -> Q4),
    Q10 = (acOff -> Q7),
    Q11 = (acOff -> Q0),
    Q12 = (acOff -> Q4).
```

Figura 4.2.16: Descrição LTS do modelo final da aplicação utilizado na ferramenta LTSA para gerar versão gráfica

Pode-se verificar o modelo final, na forma gráfica, que representa a aplicação na figura 4.2.17 a seguir.

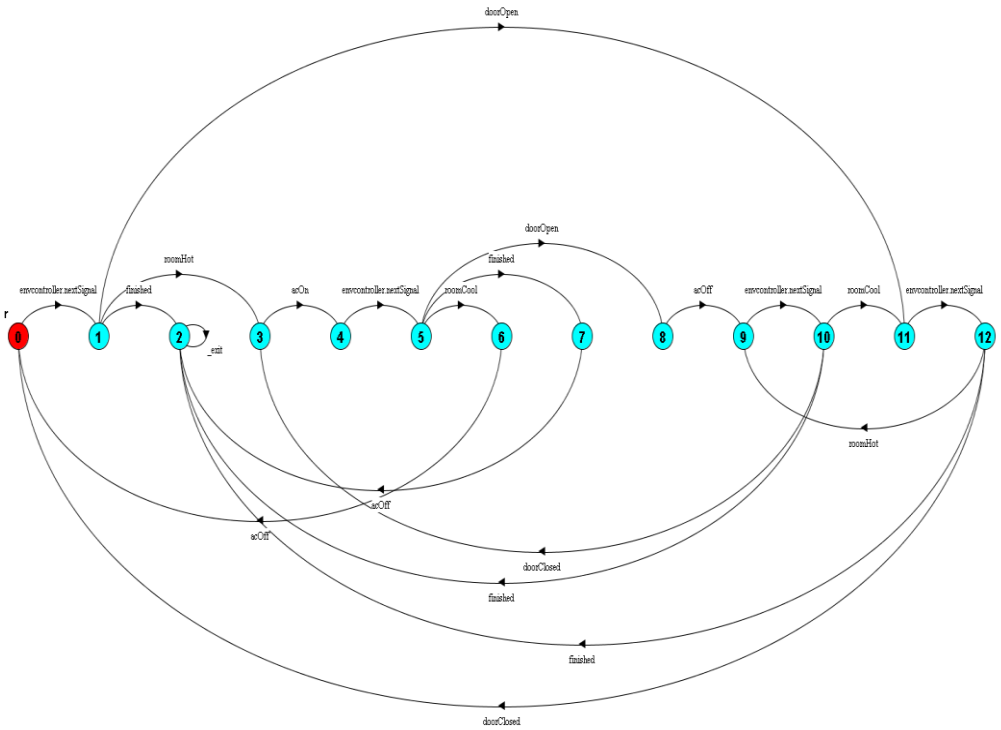

Figura 4.2.17: Modelo final da aplicação após adição de todos os testes derivados das transições negativas

Após a geração do modelo, verificaram-se as propriedades e nenhum contraexemplo ou violação foi encontrada. Assim, prosseguiu-se para os passos “14, 15 e 16” do algoritmo e a verificação contida em “16” foi satisfeita, ou seja, a lista de transições negativas é igual à lista de ações não executáveis o que implica em se ter completado a verificação exaustiva do modelo. Desta forma o algoritmo termina no passo “20” com a indicação de que o algoritmo não consegue mais identificar novos testes ou problemas no modelo em relação à especificação.

Com base nos resultados identificados na execução sobre este segundo estudo de caso, elaborou-se a seguinte tabela que descreve os resultados de forma numérica em relação à: número de testes iniciais; número de testes gerados; número de problemas identificados; número de ciclos executados do algoritmo e o número de transições no modelo final. Podem-se observar os resultados na tabela 4.2.1 a seguir:

Tabela 4.2.1: Tabela consolidada com os resultados da execução do algoritmo no segundo programa usado como estudo de caso

Número de testes iniciais	1
Número de testes válidos gerados	9
Número de testes negativos gerado	79
Número de problemas identificados/Número de erros existentes	3/3
Número de ciclos executados do algoritmo	9 + 74
Número de estados do modelo final	12
Número de transições no modelo final	21

A tabela mostra que, neste exemplo, encontraram-se todos os defeitos conhecidos da aplicação utilizada para o segundo caso de uso. Também se pode notar o aumento do número de testes gerados e do crescimento do número de transições e de estados do modelo final. Este fato se deve pela maior complexidade da aplicação utilizada como segundo estudo de caso e também pelo maior número de propriedades definidas sobre a aplicação.

4.3 Programa *MP3 Player*

Este estudo de caso foi utilizado por ser um exemplo que não se possuía conhecimento a priori sobre os mapeamentos entre os comportamentos e as entradas necessárias para se gerar os testes. Também não se tinha conhecimento do código do programa e das suas particularidades, o que em alguns casos pode gerar um teste direcionado. O código e as propriedades desde estudo de caso podem ser encontrados no Anexo C.

Este programa tem a finalidade de percorrer um diretório verificando a existência de arquivos e gerando um arquivo de saída com o nome dos arquivos. Caso encontre um arquivo do tipo *MP3,* o programa também adiciona as informações referentes às *TAGS* do arquivo *mp3.* As propriedades definem que, o programa deve ler apenas um arquivo por vez e ao terminar de ler o ultimo arquivo do diretório o programa para.

O teste inicial utilizado para o inicio do algoritmo foi: a execução do programa passando como parâmetro uma pasta que continha somente um arquivo. O modelo inicial gerado partindo deste teste pode ser visto na figura 4.3.1.

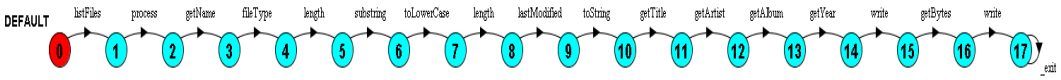

Figura 4.3.1: Modelo inicial do MP3 player gerado partindo de um teste com um diretório possuindo apenas um arquivo.

Após a geração do modelo se verificou o modelo contra as propriedades definidas e nenhuma violação foi identificada. Após este passo se transformou o modelo inicial em propriedade e tentou-se a geração de novos testes partindo dos contraexemplos, como feitos nos estudos de caso anteriores. Conseguiu-se gerar apenas um teste novo partindo do modelo partindo da sequência <*listFiles, proccess, getNames, listFiles*>. O vetor de testes necessário para gerar esta classe de equivalência foi um diretório que contivesse outro diretório internamente. Após a adição deste teste, gerou-se o modelo como pode ser visto na figura 4.3.2.

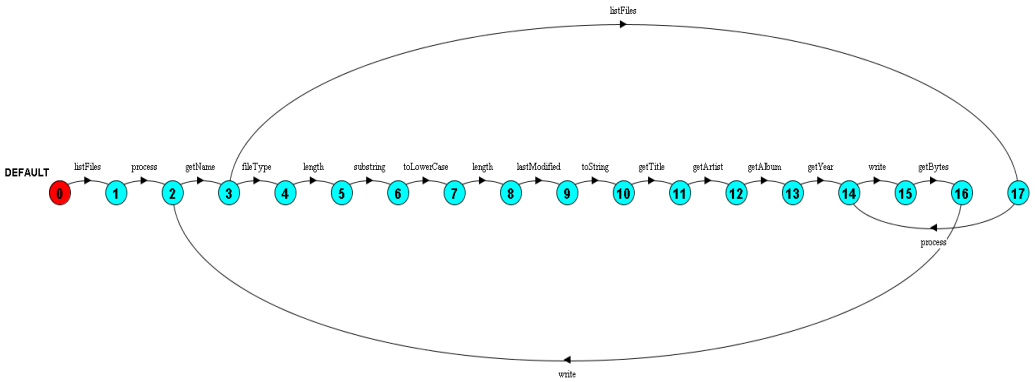

Figura 4.3.2: Modelo gerado a partir da adição de teste onde diretório possuía diretório interno.

A verificação foi feita contra as propriedades usando o novo modelo gerado e nenhuma violação foi identificada. Tentou-se gerar mais testes, mas não se conseguiu gerar classes de equivalência para nenhuma outra sequência contida na *lista de transições negativas.* Assim todas as sequências foram adicionadas a lista de ações não executáveis e o algoritmo foi encerrado.

A seguir na tabela 4.3.1 podem-se ver os resultados extraídos partindo da execução do algoritmo neste estudo de caso.

Tabela 4.3.1: Tabela consolidada com os resultados da execução do algoritmo no terceiro programa usado como estudo de caso

Número de testes iniciais	1
Número de testes válidos gerados	2
Número de testes negativos gerado	253
Número de problemas identificados/Número de erros existentes	0/0
Número de ciclos executados do algoritmo	257 [1]
Número de final de estados do modelo	17
Número de transições no modelo final	19

Como podemos ver na tabela, o número de ciclos do algoritmo cresce de forma drástica quando existe a adição de comportamentos diferentes. Pode-se ver este fato pelo número de ciclos identificados durante a geração dos testes baseado nos contraexemplos. Dentre as 19 transições, 17 destas são comportamentos distintos, assim gerando um número muito grande de testes a serem gerados partindo dos contraexemplos. Também se pode notar que a existência ou não de falhas fica diretamente ligada às propriedades definidas, pois este é o oráculo que utilizamos como referência para definir os comportamentos esperados.

4.4 Discussão

Os estudos de casos proveram uma base muito interessante de dados para sustentar a ideia de utilizar-se a integração de teste de software com verificação de modelos. Adicionalmente pode-se ver a relevância de estabelecer-se um processo parcialmente automatizado e sistemático para o ciclo de verificação e teste. A adição de testes, partindo de contraexemplos, definiu uma metodologia sistemática que permite que se faça um teste exaustivo no modelo, diferentemente de um teste exaustivo na aplicação que para programas de tamanho real torna-se muito custoso (PRESSMAN, 2005).

No primeiro estudo de caso, o editor de texto, pode-se notar que o número de ciclos presentes não apresentou um número relevante. Este fato deveu-se a que durante a execução do algoritmo, durante as primeiras fases, já se conseguiu eliminar *transições negativas* que nas fases finais do algoritmo iriam gerar mais ciclos para a sua verificação.

No segundo estudo de caso, o controlador de Ar-condicionado, pôde-se verificar a ideia de identificarem-se problemas na aplicação após a verificação do modelo gerado

[1] Destes 257 ciclos, quatro ciclos ocorreram durante a execução da primeira fase do algoritmo e 253 durante a geração de novos testes devido ao grande número de ações distintas na aplicação.

contra as propriedades. Neste estudo de caso já se viu o aumento relevante no número de ciclos necessários para a parada do algoritmo. Isto se deve ao fato de existirem maior número de ações distintas na aplicação e a presença de erros no código, pois para cada erro identificado o algoritmo teve que ser reiniciado.

No terceiro estudo de caso, o Programa de MP3, não se tinha informação a priori alguma da aplicação. Viu-se, neste caso, que a convergência do algoritmo tem uma relação forte com a definição das propriedades. Caso a especificação seja muito abrangente, o número de testes adicionais gerados e de ciclos necessários para a parada do algoritmo pode crescer bastante. Como não se tinha conhecimento do código, assumiu-se que o número de erros conhecidos era igual a zero.

Foi possível a geração de um modelo de comportamento representativo para todos os estudos de caso e conseguiram-se identificar erros na aplicação em decorrência da verificação do modelo no segundo estudo de caso. Após a correção destes erros, pôde-se verificar a correspondência da alteração nos modelos gerados, bem como a não presença das violações, quando se verificou os novos modelos corrigidos contra as propriedades.

Algumas limitações já sabidas foram identificadas durante a execução do algoritmo nos estudos de caso, tais como, o problema da escalabilidade da verificação de modelos e o problema da identificação dos atributos de refinamento. Algumas limitações adicionais foram identificadas como o aumento do número de ciclos no algoritmo quando os testes iniciais são mínimos, até que o modelo final seja gerado. Também se identificou que o processo de criação de testes partindo do modelo, ou seja, a criação das classes de equivalência para se gerar os vetores de testes para a criação dos testes gerados partindo de contraexemplos, não é um processo de fácil automatização. Mesmo manualmente este processo não é simples, pois se precisam identificar as entradas necessárias para executar determinados comportamentos.

Ainda que as limitações tenham sido identificadas, a execução do algoritmo, nestes estudos de caso e alguns casos adicionais, permitiu um aumento da confiança da utilização do algoritmo proposto para a integração de teste de software e verificação de modelos de aplicações buscando o acréscimo de cobertura de testes e a automação da verificação de propriedades, bem como a geração dos modelos partindo de rastros de execução.

4.5 Trabalhos Relacionados

Nesta seção são discutidos trabalhos relacionados que tentam se utilizar das técnicas de verificação de modelos e teste de software na busca da identificação de problemas e o aumento da cobertura do modelo e dos testes.

Em Groz et al. (2008) é utilizado um modelo do tipo *IOTS*(*Input-output Transition System*) onde as transições representam entradas e saídas que fazem o modelo trocar de estado. Esta alternativa de representação gera uma diferença na forma da geração de novos testes, pois não se pensa no comportamento, mas sim nas entradas necessárias para atingir tal estado. Diferente da abordagem utilizada neste trabalho, Groz et al. (2008) utiliza-se de uma abordagem de tentativa e erro para definir o conjunto inicial de testes para gerar o modelo para um determinado componente. Neste ponto, este trabalho não define um conjunto inicial de testes e de forma incremental aumenta-se a representatividade do modelo através da adição de informações e refinamentos do

modelo, o que acaba removendo uma restrição inicial de se ter conhecimento a priori da aplicação. Os critérios de parada utilizados são similares aos deste trabalho, porém não existe identificação de qual ação tomar quando não são possíveis mais refinamentos no modelo e ainda se identifica contraexemplo. Neste trabalho quando não se consegue mais refinar o modelo e o contraexemplo não é um comportamento da aplicação, volta-se para as propriedades a fim de verificar possível problema na especificação.

Em Walkinshaw (2009) é utilizado um modelo similar ao usado neste trabalho. Utiliza-se o modelo *PLTS* (*Partial Labelled Transaction System*), onde adicionalmente às características de um modelo *LTS* existe a definição de transições parciais que são utilizadas para diferenciar comportamentos inválidos de comportamentos desconhecidos. Nesta abordagem também é necessário um conjunto inicial de testes que contenha todo o alfabeto das propriedades. Esta restrição adiciona a necessidade do conhecimento a priori da aplicação para a geração do modelo inicial. Diferentemente, neste trabalho não se tem a necessidade de um conhecimento a priori da aplicação, pois o modelo inicial é gerado sem a necessidade de que todo alfabeto das propriedades esteja contemplado. Após a geração de um modelo inicial se aumenta a correção e a completude do modelo por passos do algoritmo. O modelo inicial de Walkinshaw (2009) possui somente um estado inicial e todas as transições adicionadas, assim a partir deste modelo são gerados testes para verificar se os comportamentos e sequências válidas no modelo são válidas na aplicação. Caso um teste falhe na aplicação esta sequência é utilizada para refinar o modelo. Diferentemente, na abordagem deste trabalho sempre se busca trabalhar com o modelo correto, assim em cada passo de iteração do algoritmo o modelo estará tão correto quanto possível.

Em Garganti (2007) é assumida a existência de um modelo inicial o qual será utilizado para derivar os testes que buscam identificar falhas. Como discutido neste trabalho, a existência a priori de um modelo não deve ser um fator restritivo para a execução de uma abordagem de testes. Diferente da abordagem proposta por Gargantini (2007), a geração do modelo neste trabalho é feita através de extração de modelos, assim tornando este processo de obtenção do modelo inicial um fator não restritivo. Para a geração de testes Garganti (2007) utiliza-se da *geração de predicados de teste* de forma automática através de uma ferramenta descrita por *Test Predicate Genarator*, assim este predicado é gerado e gera uma *Trap Property*, que é definida em PROMELA, linguagem de SPIN, e esta propriedade é dita como nunca verdadeira e busca-se um ponto onde a propriedade é tida como verdadeira para gerar o contraexemplo. Diferentemente deste trabalho, onde a geração dos testes utiliza classes de equivalência e precisa de intervenção manual, a abordagem de Garganti (2007) gera os testes de forma automática.

5 CONCLUSÃO

A proposta deste trabalho tem como principal objetivo prover, de forma automatizada, um processo para a geração de testes e identificação de problemas em sistemas informatizados. A geração de um modelo da aplicação, que possa ser utilizado pela ferramenta LSTA para verificar-se um conjunto de propriedade e que através do modelo se possa representar o comportamento da aplicação, auxilia na garantia da correção da aplicação e pode ser utilizado para aumentar a cobertura de testes a partir da geração de novos vetores de teste para a aplicação.

Diferente das abordagens utilizadas por (WALKINSHAW et al., 2009) e (GROZ et al., 2008), tentou-se utilizar o número mínimo de testes iniciais para diminuir o número de conhecimento a priori que se deve ter sobre a aplicação. Justamente por esse fato, a fase de adição de testes de completude foi uma limitação identificada, pois quanto menor o número de ações incluídas no teste inicial mais ciclos de completude são executados no algoritmo. Assim, fica evidente uma possibilidade de estudo sobre formas de identificar qual o melhor teste inicial que fará o algoritmo convergir mais rapidamente.

Outra limitação encontrada foi na heurística de seleção de atributos de refinamento para os passos de refinamento do modelo buscando a correção. Este processo ainda é manual, porém conseguiu-se definir uma estratégia para definição dos atributos. Esta limitação já foi discutida for Duarte (2007).

Ainda pode-se identificar a limitação da geração de classes de equivalência para a geração dos testes baseado nos contraexemplos. Como o modelo representa comportamentos da aplicação, não se possui necessariamente informações sobre as entradas necessárias para atingir tais comportamentos. Como o modelo utilizado possui somente representação de ações/comportamentos, sentiu-se a limitação quando se precisou fazer um mapeamento entre comportamentos a serem testados e entradas a ser utilizada para gerar este comportamento. Em trabalhos futuros poderá se tentar identificar uma heurística mais refinada para o mapeamento dos comportamentos em entradas, ou se criar um modelo que também inclua informações sobre as entradas juntamente com os comportamentos.

Uma limitação deste trabalho foram os estudos de caso utilizados. Estes estudos de caso eram apenas aplicações simples e que não continham inúmeras interdependências. Viu-se, entretanto, que a execução do algoritmo, mesmo para casos simples, ainda necessita de intervenção humana, não é completamente automático e demora um tempo não desprezível. Desta forma, deverá se buscar, em trabalhos futuros, a execução do algoritmo para mais casos, e se possível tentar provar que o algoritmo chega a um ponto de parada para a maioria dos casos relevantes. Também se poderá verificar, de forma comparativa, o tempo da execução do algoritmo em relação ao tempo individual da

execução de uma abordagem usando somente verificação de modelos e outra somente com técnicas de teste de software.

Através deste estudo viu-se a importância da definição de critérios de parada para o algoritmo, pois desta forma consegue-se chegar a um modelo que representa de forma bastante aproximada a aplicação real. Outro fator importante é o uso de abstração de informação no modelo, permitindo, desta forma, a geração de testes exaustivos no modelo que tendem a serem menos custosos do que criar estes mesmo testes de forma exaustiva na aplicação.

Como se utilizou uma técnica exaustiva no modelo para a geração dos testes, no algoritmo proposto, pode-se notar a limitação de que: caso existam muitas combinações diferentes de comportamentos na aplicação, a geração de testes partindo de contraexemplos acaba incorrendo no mesmo problema de explosão de possibilidades que temos no teste de software quando se usa uma técnica exaustiva. Desta limitação, pode-se gerar um estudo futuro para definir uma heurística buscando que a geração de testes partindo de contraexemplos não se utilize de métodos exaustivos. No algoritmo, já foram propostas algumas soluções iniciais como: adicionar sequências não executáveis identificadas à *lista de ações não executáveis* antes do início da fase de geração do teste partindo de contraexemplos, o que acarreta na diminuição do número de possibilidades a ser geradas posteriormente. Porém, fica claro que esta solução não é a única muito menos a ideal.

A partir dos resultados encontrados, viu-se que o algoritmo consegue identificar falhas na aplicação após a verificação do modelo contra as propriedades definidas. Através da geração do modelo de forma automática, se diminui o custo da geração do modelo e se adiciona o fato das informações utilizadas no modelo serem reais, assim buscando a não adição de comportamentos “não existentes” ao modelo por erro na modelagem.

Durante a execução do algoritmo sempre se buscou garantir a correção e a completude do modelo, assim quando se geraram novos testes baseados no modelo, sabia-se que qualquer erro gerado é relativo aos novos comportamentos adicionados.

Finalmente, poderá ser feito um estudo sobre os possíveis impactos sobre os artefatos existentes, como modelo final, lista de testes, lista de ações não executáveis, quando for feita alguma alteração no programa em seu código fonte. Poderá ser extraída uma métrica sobre o reuso dos artefatos e uma heurística para reaproveita-los durante a execução do algoritmo novamente, utilizando uma ideia de executar novamente os testes para garantir que comportamentos identificados e não alterados continuem válidos.

REFERÊNCIAS

CLARKE, E.M.; WING, J. M.. Formal Methods: State of Art and Future Directions. **ACM Computing Survey**: 28(4):p. 626-643, 1996.

CLARKE, E.M.; GRUMBERG, O.; PELED, A.D.. **Model Checking.** The MIT Press, Cambrigde, Massachusets, EUA, 1999.

GROZ, R.; LI, K; PETRENKO, A.; SHAHBAZ, M.; **Modular System Verification by inference, Testing and Reachability Analysis**. TestCom/FATES 2008, LNCS 5047, pp. 216–233, 2008.

MANNA, Z.; PNUELI, A.. **The Temporal Logic of Reactive and Concurrent Systems.** Springer-Verlag New York, Inc., Nova Iorque, NY, EUA, 1992.

HOLZMANN, G.J.. From Code to Models. **In ACSD**, pages 3–10, Newcastle upon Tyne, UK, June 2001.

WALKINSHAW, N; DERRICK, J.; GUO, Q.. **Iterative Refinement of Reverse-Engineered Models by Model-Based Testing**. Department of Computer Science, The University of Sheffield, Sheffield, UK, 2009.

MAGEE, J.;KRAMMER, J.. **Concurrency: State Models and Java Programming**, 2 ed., Wiley and Sons, 2006.

DUARTE, L.M.. **Behaviour Model Extraction using Context Information**. 2007. 245 f. Dissertação (Doutorado em Ciência da computação) - University of London and the Diploma of Imperial College.

HOLZMANN, G.J.; SMITH, M.H.. **A Practical Method for Verifying Event-DrivenSoftware**. Em Conferência de Engenharia de Software, p. 597–607, Los Angeles, EUA, 1999.

LEUSCHEL, M.; MASSART, T.; CURRIE, A.. **How to Make FDR Spin: LTL Model Checking of CSP by Refinement**. Notas de palestra em Ciência da Computação, 2021:99–118, Março 2001.

GIANNAKOPOULOU, D.; MAGEE, J.. Fluent Model Checking for Event-Based Systems. **ESEC/FSE**, p. 257–266, Helsinki, Finlandia, Setembro 2003.

MAGEE, J.; KRAMER, J.; CHATLEY, R.; UCHITEL, S.; FOSTER, H.. **LTSA - Labelled Transition System Analyser**. Disponível em: <http://www.doc.ic.ac.uk/ltsa/> acessado em 07/05/2012.

PRESSMAN, R.S.. **Software Engineering: A Practitioner's Approach**, 5 ed., McGraw - Hill Science/Engineering/Math, p. 437-503, 2005.

INTERNATIONAL SOFTWARE QUALIFICATION BOARD, **ISTQB 2011.** Certified Tester Foundation Level Syllabus. Abril, 2011. Disponível em <http://www.istqb.org/downloads/finish/16/15.html> acessado em 08/05/2012.

BERTOLINO, A.. Software Testing Research: Achievements, Challenges, Dreams. **Future of Software Engineering(FOSE'07),** IEEE, Istituto di Scienza e Tecnologie dell'Informazione, Pisa, Italia, 2007.

GARGANTINI, A.. **Using Model Checking to Generate Fault Detecting Tests.** Departament of Management and Information Technology**,** Università di Bergamo, 2007.

BEYER, D.; CHLIPALA, A.J.; HENZINGER, T.A; JHALA, R.; MAJUMDAR, R.. Generating Tests from Counterexamples, **Proceedings...**(ICSE'04), IEEE, 2004.

PATTON, R.. **Software Testing**. Sams, 2 ed. , 2006.

UCHITEL S.; KRAMER J.; MAGEE J.. **Behaviour Model Elaboration Using Partial Labelled Transition Systems.** ESEC/FSE, p. 19–27, Helsinki, Finland, Setembro, 2003.

AHO A.; SETHI R.; ULLMAN J.. **Compilers: Principles, Techniques and Tools.** Addison-Wesley, 1986.

COOK J. E.; WOLF A. L.. **Discovering Models of Software Processes from Event-Based Data.** ACM Transactions on Software Engineering and Methodology, p . 215–249, Julho, 1998.

AMMONS G., BODIKÌK R.;LARUS J. R.. Mining Specifications. **ACM, Simpósio sobre Principios de Linguagens de Programação**, p. 4–16, Portland, EUA, Janeiro, 2002.

ERNST M. D.. **Static and Dynamic Analysis: Synergy and Duality**. Workshop em Dynamic Analysis, p. 24–27, Portland, EUA, Maio, 2003.

ANEXO A <CÓDIGO E PROPRIEDADES EDITOR DE TEXTO>

Propriedades em FLTL para o editor de texto:

```
// P1
property OpenAndClose = CLOSED,
CLOSED = (open -> OPEN),
OPEN = (close -> CLOSED).

// P2
property SaveOnlyIfEdited = SAVED,
SAVED = (edit -> EDITED),
EDITED = (edit -> EDITED
        |save -> SAVED).

// P3
fluent Open = <open, close> initially 0
assert NotAllowed = (!Open -> !(edit || print || save))
```

Código do editor de texto:

```
import java.io.*;
/**
 * Simulates the command reader of the editor system.
 *
 * @author Lucio Mauro Duarte
 * @version 10/04/08
 */
class CommandReader {
        BufferedReader r = null;
        public CommandReader (String inputFile) throws IOException {
                r = new BufferedReader (new FileReader (inputFile));
        }
        public CommandReader () throws IOException {
                r = new BufferedReader (new InputStreamReader (System.in));
        }
        protected String readCommand () throws IOException {
```

```
                        return r.readLine ();
                }
        }
/**
  * Implements the editor system.
  *
  * @author Lucio Mauro Duarte
                * @version 10/04/08
  */
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: ");
       cmd = Integer.parseInt (r.readCommand ());
      } 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 = Integer.parseInt (r.readCommand ());
     if (opt == 0)
            save ();
    } catch (Exception e) {}
                        }

  if (isOpen)
            close ();
```

```
  }
}

import java.io.IOException;

/**
 * Main class of the editor system.
 *
 * @author Lucio Mauro Duarte
 * @version 10/04/08
 *
 */
public class EditorMain {
	public static void main (String args[]) {
		try {
			new Editor (new CommandReader ());
		}
		catch (IOException ex) {
			System.out.println ("*** Error executing editor! ***");
		}
	}
}
```

ANEXO B <CÓDIGO E PROPRIEDADES AR-CONDICIONADO>

Propriedades em FLTL ar condicionado:

```
fluent AC_ON = <acOn, acOff> initially 0
fluent DOOR_CLOSED = <doorClosed,doorOpen> initially 1
fluent ROOM_HOT = <roomHot, roomCool> initially 0
fluent SYSTEM_OFF = <finished,envcontroller.nextSignal> initially 0

// If room is hot and door is closed, then turn AC on
assert AC_ACTIVE = [](ROOM_HOT && DOOR_CLOSED && !SYSTEM_OFF) -> X AC_ON)
// If room is cool or door is open, then turn AC off
assert SAVE_ENERGY = [](!ROOM_HOT || !DOOR_CLOSED) -> X !AC_ON)
// If system is exiting, AC must be turned off
assert AC_OFF = [](SYSTEM_OFF && AC_ON) -> X !AC_ON)

assert CORRECT_AC = (AC_ACTIVE && SAVE_ENERGY && AC_OFF)
```

Código ar-condicionado:

```
/**
 * Defines the signals exchanged between the controller and the
 * air conditioner system.
 *
 * @author particular
 *
 */
public interface Signals {
        static final int ROOM_HOT                = 0;
  static final int ROOM_COOL                     = 1;
  static final int DOOR_OPEN                     = 2;
  static final int DOOR_CLOSED        = 3;
  static final int OFF                                                  = 4;
}
```

```
import java.io.*;

/**
 * Simulates the environment controller of the air conditioner system.
 *
 * @author Lucio Mauro Duarte
 * @version 11/04/08
 */

class EnvController {
	private BufferedReader in;

	public EnvController () {
		in = new BufferedReader (new InputStreamReader (System.in));
	}

	public int nextSignal () {
		int signal = -1;

		try {
			signal = Integer.parseInt (in.readLine ());
		}
		catch (Exception e) {}

		return signal;
	}
}
/**
 * Main class of the air conditioner system.
 *
 * @author Lucio Mauro Duarte
 * @version 11/04/08
 */

class ACController {
	public static void main (String args []) {
	EnvController c = new EnvController ();
	new AirConditioner (c);
	}
}

/**
 * Implements the air conditioner system controller.
 *
 * @author Lucio Mauro Duarte
```

```
 * @version 11/04/08
 */

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;
                        #action:"roomHot";
                        System.out.println ("-> Room hot");
                        }
                        break;

        case ROOM_COOL:  if (room_hot) {
                        room_hot = false;
                        #action:"roomCool";
                        System.out.println ("-> Room cool");
                        }
                        break;

        case DOOR_OPEN: if (door_closed) {
                                door_closed = false;
                                #action:"doorOpen";
                                System.out.println ("-> Door open");

                                if (ac_on) {
                                ac_on = false;
                                #action:"acOff";
                                System.out.println ("-> AC off");
                                }
```

```
        }
        break;

        case DOOR_CLOSED: if (!door_closed) {

        door_closed = true;

        #action:"doorClosed";

        System.out.println ("->Door closed");

        if (room_hot) {
        ac_on = true;

        #action:"acOn";

        System.out.println ("-> AC on");
        }
        }
    break;

        case OFF:  finished = true;
                                                        #action:"finished";
                                                        break;

        default: System.out.println ("Incorrect command!");
        }
                }
        }
}
```

ANEXO C <CÓDIGO E PROPRIEDADES MP3 PLAYER>

Propriedades do sistema:

```
//P1: A cada instante, o sistema deve processar apenas um arquivo.
assert P1 = [](arqRec -> !X(arqRec))

//P2: O sistema para após ler o último arquivo do diretório.
assert P2 =[](fimLista -> !X(verificaArq || verificaArqNotMusic))
```

Código do programa:

```
package MP3Tag;

import java.io.*;



public class FileMP3Tag {

   // Declaração das variáveis
   // Neste caso declarei-as como private
   // para fazer o chamado "encapsulamento"
   // que nada mais é do que permitir o acesso
   // aos atributos por meio dos métodos da classe

      private String title;
   private String artist;
   private String album;
   private String year;

   // Construtor da classe
   // Normalmente é o construtor que inicializa
   // as variáveis da classe
   public FileMP3Tag( File fileMP3 ) {
                  try {
                                 FileInputStream file = new FileInputStream( fileMP3 );
                                 int size = (int) fileMP3.length();
                                 file.skip(size - 128);
```

```
                        byte[] last128 = new byte[128];
                        file.read(last128);
                        String id3 = new String(last128);
                        String tag = id3.substring(0, 3);
                        if (tag.equals("TAG")) {
                                this.title = id3.substring( 3, 32);
                                this.title = title.trim();
                                this.artist = id3.substring(33, 62);
                                this.artist = artist.trim();
                                this.album = id3.substring(63, 91);
                                this.album = album.trim();
                                this.year = id3.substring(93, 97);
                                this.year = year.trim();
                        } else {
                                System.out.println(fileMP3.getName() + " does not contain" + " ID3 info.");
                        }
                        file.close();
                } catch (Exception e) {
                        System.out.println("Error -- " + e.toString());
                }

    }

    // Daqui em diante ficam os getters e setters
    // que são os métodos que fazem o acesso
    // (gravação ou leitura) dos atributos da classe
    public String getTitle() {
        return this.title;
    }

    public String getArtist() {
        return this.artist;
    }
    public String getAlbum() {
        return this.album;
    }
    public String getYear() {
        return this.year;
    }

}
```