Verificação Automática de Programas a partir da Monitoração de Múltiplas Execuções de Código

Genildo de Moura Vasconcelos

Dissertação de Mestrado submetida à Coordenação do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Angelo Perkusich, D.Sc.
Orientador
Leandro Dias da Silva, D.Sc.
Orientador

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

V331v Vasconcelos, Genildo de Moura.
Verificação automática de programas a partir da monitoração de múltiplas execuções de código/Genildo de Moura Vasconcelos – Campina Grande, 2012.
51f.: il.

Dissertação (Mestrado em Engenharia Elétrica) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientadores: Prof. Dr. Angelo Perkusich, Profo. Dr. Leandro Dias da Silva.
Referências.

1. Engenharia de Software. 2. Métodos para Verificação de Programas. 3. Execuções de Código. I. Título.

CDU 004.41(043)

VERIFICAÇÃO AUTOMÁTICA DE PROGRAMAS A PARTIR DA MONITORAÇÃO DE MÚLTIPLAS EXECUÇÕES DE CÓDIGO

GENILDO DE MOURA VASCONCELOS

Dissertação Aprovada em 16.03.2012

ANGELO PERKUSICH, D.Sc., UFCG
Orientador

LEANDRO DIAS DA SILVA, D.Sc., UFAL
Orientador

ANTONIO MARCUS NOGUEIRA LIMA, Dr., UFCG
Componente da Banca

KYLLER COSTA GORGÔNIO, D.Sc., UFCG
Componente da Banca

CAMPINA GRANDE - PB
MARÇO - 2012

À Ana Emília, Ângela, José Batista, Gheysa e Geliane.

Agradecimentos

Ao Criador do Universo, a grande força que ilumina minhas escolhas e minha vida.

Aos meus pais por todo o apoio, sempre foram exemplo de força e superação.

À minha amada esposa Ana Emília Holanda Rolim pela paciência e incentivos em todos os momentos.

Aos meus orientadores Angelo Perkusich e Leandro Dias pela atenção, honestidade, competência e profissionalismo no desempenho do trabalho. Agradeço também à copele e à Angela pelo suporte durante todo o curso.

Aos grandes amigos, Santana, Saulo, Diogo, Diego, Tiago Onofre, Yuri C. Gomes, Allan Jones, Zé Luis, Danilo, Jaidilson, Jadsonlee, David, Tomás, Fabrício, Patrício, Ademar (Shurato) e Marcelo Amorim por terem contribuído com a minha formação.

À Ford Motor Company, em especial Eude Oliveira, Gustavo Schiavotelo, Flávia Araújo e Cláudio Moles, pelos incentivos ao meu desenvolvimento.

Resumo

O objetivo neste trabalho é apresentar um método para a verificação de programas a partir de modelos gerados com a análise de múltiplos traços de execuções do código. Para tanto, o programa a ser verificado deverá ser instrumentado para que ocorra a coleta dos traços das execuções em um arquivo de *log*. A partir desse arquivo, um algoritmo é aplicado para a obtenção de um modelo reduzido do programa. Esse modelo será então confrontado com a especificação do produto por meio de um verificador de modelos para a detecção de contra-exemplos da especificação. Os contra-exemplos detectados representam erros presentes no código fonte do programa implementado. Estes erros deverão ser corrigidos e as etapas repetidas para uma nova verificação. A validação desse método de verificação automática de programas é apresentado em um estudo de caso.

Abstract

The objective in this work is to present a method for verifying programs using models generated with the analysis of multiple execution traces. The program to be checked should be instrumented to obtain the traces information and recording in a log file. Using this file, an algorithm is applied to obtain a reduced program model. This model is then confronted with the product specification through a model checker to detect counter-examples of the specification. The counter-examples detected are caused due source code errors of the program implemented. These erros should be corrected and steps repeated to re-check the specification. The validation of this method for automatic formal verification of software system is presented as a case study.

Sumário

Glossário

LTL	*Linear Temporal Logic*
CTL	*Computation Tree Logic*
API	*Application Programming Interface*
MEF	*Máquina de Estados Finito*
EEF	*Espaço de Estados Finito*
ABS	*Anti-lock Braking System*
NuSMV	*New Symbolic Model Verifier*
SMV	*Symbolic Model Verifier*

Lista de Figuras

Capítulo 1

Introdução

O desenvolvimento dos sistemas embarcados nas últimas décadas trouxe uma forte expansão no número de aplicações dependentes de dispositivos programáveis. Hoje em dia, existem dispositivos programáveis aplicados em áreas tão distintas como automobilística; médica; sistemas financeiros; entre outras. Uma eventual falha em algum desses sistemas pode provocar diferentes graus de danos e prejuízos, por isso, exige-se um alto grau de confiança no funcionamento para todos eles. Os sistemas de controle críticos, como por exemplo, o freio com sistema anti-travamento em veículos (*Anti-lock Braking System - ABS*) e os equipamentos de monitoramento médicos, podem colocar vidas humanas em grave risco, caso haja a ocorrência de falhas. As possíveis falhas, em sistemas que controlam a geração e distribuição de energia, ou ainda, em sistemas telefônicos, podem provocar enormes transtornos à sociedade. Mesmo os sistemas de controle que não ameaçam diretamente a integridade humana, devem possuir um alto grau de confiança no funcionamento, pelo fato de que eventuais falhas podem resultar em graves transtornos econômicos para as companhias envolvidas. Além disso, a necessidade de um alto grau de confiança no funcionamento dos dispositivos é acompanhando pelo aumento da complexidade dos novos sistemas, ao mesmo tempo em que, por pressão econômica, os prazos de entrega dos dispositivos programáveis se mantêm os mesmos ou menores. Tanto o aumento de complexidade quanto o diminuição no prazo de entrega tornam bem mais difícil a tarefa da verificação e validação do sistema a ser entregue, e conseqüentemente, exigem que maiores esforços sejam despendidos (1).

O processo de teste de software é a abordagem mais freqüentemente aplicada na indústria durante o processo de verificação e validação do software dos dispositivos programáveis, podendo este processo ser considerado como a última revisão da especificação e implementação do sistema durante o desenvolvimento do produto (2). Atualmente, estima-se que 60% dos custos durante o processo de desenvolvimento de software, sejam destinados ao processo de teste do software (1,3). Assim, os altos custos e a necessidade

de um alto grau de confiança no funcionamento dos dispositivos programáveis, proporcionam um grande estímulo à pesquisa em busca de alternativas de verificação e validação do software dos dispositivos programáveis.

Neste contexto, técnicas de teste baseadas em modelos, são estudadas como uma alternativa eficaz para serem aplicadas durante o processo de verificação e validação do software (4, 5). Nestas técnicas, usa-se uma notação formal para modelar o sistema e a especificação sob teste, e uma ferramenta de verificação de modelos irá analisar automaticamente se o modelo atende ou viola a especificação.

Infelizmente, o processo de notação formal para a obtenção do modelo requer um conhecimento especial na linguagem da ferramenta de modelagem e nos fundamentos teóricos da verificação de modelos. E mesmo com o conhecimento necessário, a criação e manutenção dos modelos não é uma tarefa trivial (6). Assim, o tempo e custo demandado pela técnica de teste baseada em modelos pode torna-se uma alternativa inviável para a indústria (7). Dessa forma, a automatização desse processo é desejada, facilitando a aplicação de uma técnica de verificação formal em programas que não haviam sido submetidos a técnicas similares anteriormente.

Neste trabalho, propõe-se um método para a verificação de software a partir do modelo reduzido do programa construído automaticamente com informações de múltiplos traços de execuções do programa. A partir do modelo obtido, uma ferramenta de verificação de modelos implementará uma varredura exaustiva. Essa varredura tem por objetivo encontrar um contra-exemplo que negue a especificação. Caso, nenhum contra-exemplo seja obtido, terá sido provado que a especificação é verdadeira. Caso contrário, o contra-exemplo encontrado será utilizado para auxiliar na análise de problemas no código fonte do programa. Para ilustrar a aplicação da técnica proposta, será apresentado um estudo de caso não trivial na área automobilística, sendo esta área escolhida devido ao crescente número de aplicações embarcadas (8).

Revisão Bibliográfica

Referente às técnicas de verificação e validação de software baseadas em informações coletadas dos traços das execuções do programa, podemos destacar os seguintes trabalhos.

Geilen (9) apresenta um método para a verificação de programas a partir da construção de tableau para a análise de traços de execuções. Uma construção de tableau é um algoritmo que transforma uma fórmula de lógica temporal em um autômato de estados finitos que aceita precisamente todos os modelos da fórmula. Esta construção é um ingrediente chave para a verificação da validade de uma fórmula, bem como para a abordagem teórica de autômatos para a verificação do modelo.

Em outra abordagem proposta por Meyer (10), o programador pode inserir asserções no programa. Embora não ocorra a utilização explícita dos traços do programa, existe a utilização implícita, visto que cada estado do traço deve respeitar a asserção introduzida. Idealmente, a sintaxe das asserções é próxima da linguagem de programação utilizada, portanto, de fácil aplicação prática pelos programadores. As asserções são verificadas durante a execução do programa e, dessa forma, a violação é detectada em tempo real. Já Bartetzko (11), estendeu para a linguagem Java o método proposto por Meyer (10), o que permitiu analisar programas Java com as especificações na forma de asserções.

Na análise de múltiplos traços, destacam-se alguns trabalhos que consideraram o problema da verificação de programas como algo relevante a ser analisado.

Ammons et al. (12) apresentam a noção de extração de especificação (*specification mining*), uma abordagem para descobrir protocolos a partir de traços de execuções de programas. A abordagem emprega a instrumentação de código e o estudo de autômatos. Com propósitos similares, Lo et al. (13) investigam uma técnica para extrair de traços de execuções do programa, um conjunto completo de fórmulas de lógica temporal de comprimentos arbitrários. A extração das regras temporais revelam quais invariantes são detectadas no programa, e conseqüentemente irá guiar os desenvolvedores a entender o seu comportamento e com isso, facilitará a verificação do mesmo.

Ernst et al. (14) introduzem Daikon, uma máquina de inferência de invariantes usadas para inferir prováveis invariantes nos traços de execuções dos programas. Estas invariantes são descritas como invariantes prováveis, pois são incluídas em todas as observações dos traços, que podem ou não conter uma amostra representativa do comportamento do sistema. Exemplos de invariantes incluem, por exemplo, $x < 100$ (o valor de x é sempre observado como menor do que 100), e $x \neq 200$ (o valor de x é observado como sempre diferente de 200). Muitas técnicas de análise de traços de execuções e geração de modelos fazem uso de Daikon. Em geral, as invariantes inferidas podem ser observadas como propriedades que são desejadas em certos pontos do sistema sob análise.

Técnicas de testes baseadas em invariantes inferidas nos programas também incluem Agitator (15), Eclat (16) e a técnica proposta por Xie e Notkin (17). Cada uma dessas abordagens, apresenta uma ferramenta que gera os dados de entrada de teste, e baseado na captura dos traços das execuções, apresenta um conjunto de invariantes descrevendo o comportamento do sistema. Com esse conjunto de invariantes, pode-se analisar se o sistema está funcionando conforme especificado, e tornar as invariantes em asserções para formar novos casos de teste.

Kanstrén et. al. (7) apresentam uma técnica para gerar automaticamente um modelo inicial que descreve o sistema a partir de traços de execução e Daikon (14), usando uma notação de linguagem de programação comum. A partir desse modelo inicial é criado

um modelo completo para ser usado para testes baseados em modelos do sistema em teste. Essa abordagem requer um menor esforço em relação à escrita de um modelo sem a aplicação da técnica. De forma similar, Lorenzoli et al. (18) modela um sistema com base em traços capturados. Nessa abordagem, utiliza-se máquinas de estados finitos e Daikon (14) para a criação do modelo. Este modelo é utilizado para a otimização da seleção dos casos de testes.

O presente trabalho não tem como objetivo extrair as invariantes dos traços. No entanto, busca-se a construção de um modelo reduzido do programa implementado com as informações coletadas nos traços das execuções do programa.

A verificação de modelos é tradicionalmente empregada como um dos métodos de verificação exaustivo, que demonstra formalmente a presença ou ausência de alguma propriedade sobre um sistema especificado. Entretanto, isto também resulta em problemas de eficiência, que motiva o desenvolvimento de algoritmos aproximados de verificação de modelos, em que apenas uma parte dos possíveis caminhos de execução são explorados. Tais algoritmos são estudados e propostos nos trabalhos de Cho et al. (19) e Salem et al. (20) e que foram utilizados como base para o presente trabalho.

Objetivos

Mediante a necessidade de um alto grau de confiança no funcionamento do programa dos dispositivos programáveis, neste trabalho tem-se como objetivo, apresentar e validar um método para a verificação automática de programas, a partir da verificação formal do modelo gerado com informações de múltiplos traços.

Os objetivos específicos desse trabalho são os seguintes:

- Definir e detalhar um método para a verificação automática de programas.
- Apresentar um estudo de caso com a finalidade de demonstrar a aplicação do método proposto.

Organização do Texto

Este documento está estruturado da seguinte forma: No Capítulo 2 observam-se os fundamentos teóricos com os conceitos básicos necessários para o entendimento da técnica de verificação de modelos. No Capítulo 3, é apresentado o método proposto para a verificação de modelos a partir da análise de múltiplos traços. No Capítulo 4, um estudo de caso é apresentado para demonstrar a aplicação do método proposto. No Capítulo 5 apresentam-se as conclusões e as perspectivas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo apresenta-se o embasamento teórico necessário para o entendimento deste trabalho. Inicialmente, apresenta-se a notação que será utilizada neste trabalho para, em seguida, apresentar os conceitos de autômato, lógicas temporais e verificação de modelos.

2.1 Notação

As seguintes regras e definições de notação serão utilizas neste trabalho:

- Um *alfabeto* é um conjunto finito e não vazio de símbolos. O símbolo $\sum$ representa o alfabeto utilizado tanto por linguagens de palavras finitas quanto por linguagens de palavras infinitas;
- Uma *linguagem de palavras finitas* é um conjunto de seqüências finitas de símbolos pertencentes a um alfabeto de entrada;
- Uma *linguagem de palavras infinitas* é um conjunto de seqüências infinitas de símbolos pertencentes a um alfabeto de entrada;
- No contexto do trabalho, os termos *traço*, *cadeia*, *seqüência* e *palavra* são intercambiáveis;
- O conjunto dos números naturais é representado por $\mathbb{N}$;
- O conjunto dos números inteiros positivos é representado por $\mathbb{N}^+$;
- Uma palavra finita é representada pelas letras u e v. O $(i+1)$-ésimo elemento de uma cadeia finita u é representado por u_i, sendo i um inteiro não negativo;
- Uma palavra infinita é representada pela letra σ. O $(i+1)$-ésimo elemento de uma cadeia infinita σ é representada por σ_i;

- O comprimento $|u|$ de uma palavra finita $u = u_0u_1...u_n$, corresponde ao número de símbolos presentes em u e é tal que $|u| = n + 1$.

2.2 Estruturas de Kripke

Seja um sistema que possa ser modelado por um conjunto finito de estados e relações de transição entre estados. Dado um conjunto finito de propriedades associadas ao modelo, é possível definir, para cada um dos estados, um subconjunto das propriedades que sejam verdadeiras naquele estado. Estas propriedades são denominadas proposições atômicas e definem cada um dos estados de maneira única, já que dois estados diferentes não podem satisfazer exatamente o mesmo conjunto de proposições atômicas. Caso obedecessem seriam indistinguíveis, ao menos até o ponto em que as proposições atômicas do problema conseguem descrevê-los.

Uma estrutura de Kripke é uma construção baseada em estados e transições para qual cada estado está associado a um subconjunto de proposições atômicas. Formalmente, uma estrutura de Kripke sobre um conjunto P de proposições atômicas $\{p_0, ..., p_{n-1}\}$ é uma quádrupla $M = (S, S_0, R, \lambda)$ tal que:

- S é um conjunto finito de estados.
- $S_0 \subseteq S$ é o conjunto de estados iniciais.
- $R \subseteq S \times S$ é a relação total de transição de estados. Isto significa que, para casa $s \in S$, existe um $s' \in S$ tal que $(s, s') \in R$.
- $\lambda : S \rightarrow 2^P$ é a função que rotula cada estado $s \in S$ com os valores 1 ou 0 associados a cada uma das $p_i \in P$. O valor associado é 1 caso p_i seja verdadeira e 0 caso contrário.

Uma trajetória sobre a estrutura M a partir de um estados $s_0 \in S_0$ é definida como uma seqüência infinita de estados $\sigma = s_0s_1...s_n$ tal que $s_0, s_1, ..., sn \in S$ e $R(s_i, s_{i+1})$ seja definida para todo $i \geq 0$. Uma seqüência de rótulos de σ é a palavra infinita $\lambda(s_0)\lambda(s_1)...$ sobre o alfabeto $\{0, 1\}^{|P|}$.

Nota-se que, de acordo com a definição acima, cada proposição atômica corresponde a um valor binário. Isto nem sempre corresponde à melhor representação para os modelos de sistemas reais. Seja, por exemplo, um sistema de controle de um forno industrial em que a temperatura seja uma variável de interesse. Poderia haver o interesse em dividir a temperatura em três faixas distintas: abaixo de 1000°C, entre 1000°C e 1200°C e acima de 1200°C. Assim, a temperatura poderia ser representada pela letra F (frio) quando estivesse na primeira faixa, N (normal) quando estivesse na segunda e Q (quente) quando

estivesse na terceira. Neste caso, a variável associada à temperatura não poderia mais ser considerada um proposição atômica como definida acima, por possuir três estados possíveis. Entretanto, este tipo de problema não limita a definição de estrutura de Kripke, já que é possível definir duas novas proposições atômicas a e b associadas aos três estados possíveis. Com isto, por exemplo, a poderia ser verdadeira quando a temperatura estivesse abaixo de 1000°C e falsa nas outras ocasiões, enquanto b poderia ser verdadeira se a temperatura estivesse entre 1000°C e 1200°C e falsa nas outras ocasiões. Se este procedimento for adotado, é possível obedecer à definição acima ao custo de um aumento na complexidade da representação do sistema.

2.3 Lógicas Temporais

Lógicas temporais são formalismos que permitem descrever e analisar os comportamentos de estruturas de Kripke a partir do comportamento das proposições atômicas ao longo das trajetórias. A preocupação é descrever as seqüências para as quais proposições atômicas são satisfeitas à medida que as possíveis trajetórias evoluem.

As lógicas temporais mais utilizadas para verificação de modelos baseados em máquinas de estados finitos são as lógicas LTL e CTL. Segue uma breve introdução de cada uma delas.

2.3.1 Lógica temporal LTL

Na lógica temporal LTL (*Linear Temporal Logic*) utilizam-se proposições atômicas, operadores booleanos e operadores temporais para construir fórmulas que especificam propriedades de cada uma das possíveis trajetórias das estruturas de Kripke.

A semântica dos operadores temporais é definida a partir de uma trajetória específica. Embora muitos operadores temporais possam ser definidos, os operadores mais comuns são $\mathbf{F}f$(a fórmula f será válida em algum futuro), $\mathbf{G}f$(a fórmula f é válida globalmente - é válida e sempre será válida), $f_1 \mathbf{U} f_2$ (*until* - f_1 é válida até que f_2 seja válida, o que acorrerá inevitavelmente) e $\mathbf{X}f$(*next* - f será válida no próximo estado).

Formalmente, é possível definir a sintaxe de uma fórmula pertencente ao conjunto de fórmulas da lógica LTL sobre um conjunto P de proposições atômicas a partir das seguintes regras:

1. Qualquer proposição atômica $p \in P$ é uma fórmula LTL.
2. Se f_1 e f_2 são fórmulas LTL, $f_1 \wedge f_2$ e $\neg f_1$ são fórmulas LTL.
3. Se f_1 e f_2 são fórmulas LTL, $f_1 \mathbf{U} f_2$ e $\mathbf{X} f_1$ são fórmulas LTL.

Pode-se definir outros operadores booleanos a partir das definições acima. As possíveis novas definições são as seguintes:

- $f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$
- $f_1 \rightarrow f_2 \equiv \neg f_1 \vee \neg f_2$
- $\top \equiv \neg f_1 \vee f_1$

Nota-se que os operadores **F** e **G** podem ser definidos a partir do operador **U** através das equivalências $\mathbf{F}f_1 \equiv (\top \mathbf{U} f_1)$ e $\mathbf{G}f_1 \equiv \neg\mathbf{F}\neg f_1$. Na Figura 2.1 apresenta-se, de forma gráfica, seqüências de estados que satisfazem a algumas fórmulas LTL básicas.

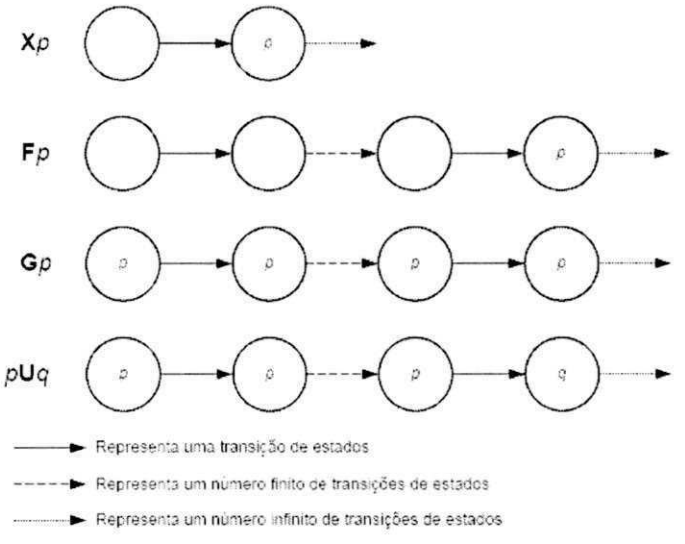

Figura 2.1: Seqüencia de estados para fórmulas LTL básicas

2.3.2 Lógica temporal CTL

As fórmulas LTL são aplicáveis a trajetórias infinitas individuais de uma estrutura de Kripke. As fórmulas CTL (*Computing Tree Logic*), por sua vez, aplicam-se a estados de uma estrutura de Kripke. Estas fórmulas são construídas associando-se os quantificadores **E** e **A** aos operados temporais da lógica LTL. Dado qualquer operador temporal Θ, o quantificador **E** torna a sub-fórmula CTL $\mathbf{E}\Theta f$ verdadeira sempre que existir alguma trajetória a partir do primeiro estado de σ tal que Θf seja verdadeiro. O quantificador **A**, por sua vez, torna a sub-fórmula $\mathbf{A}\Theta f$ verdadeira sempre que todas as possíveis trajetórias a partir do primeiro estado de σ obedeçam a Θf.

A partir de um conjunto P de proposições atômicas, é possível definir a sintaxe de uma fórmula CTL a partir das regras abaixo:

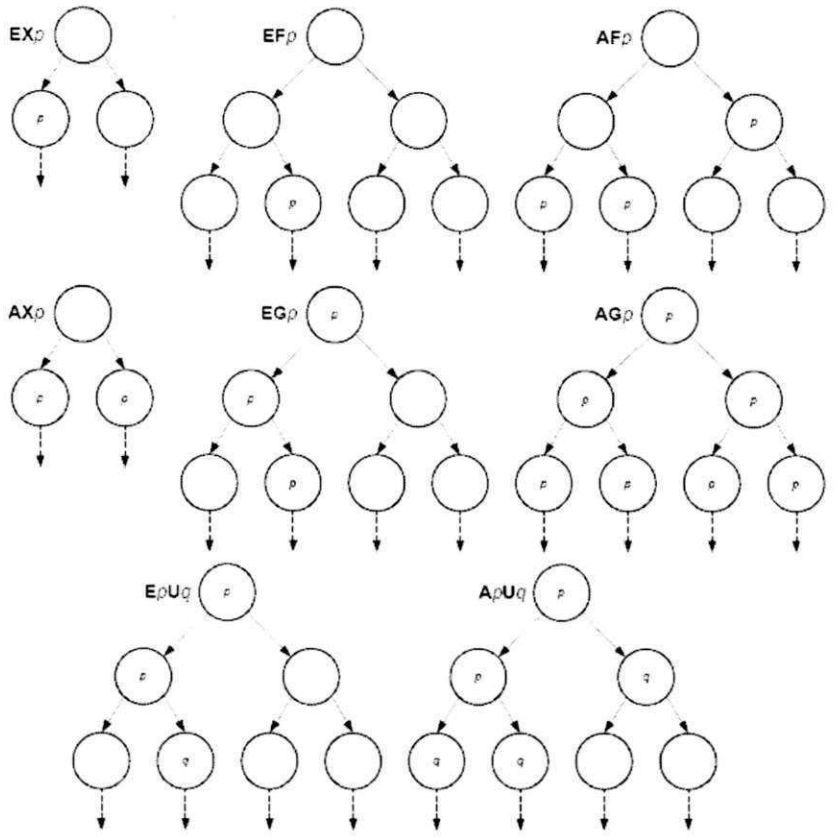

Figura 2.2: Seqüencia de estados para fórmulas CTL básicas

1. Qualquer proposição atômica $p \in P$ é uma fórmula CTL.
2. Se f_1 e f_2 são fórmulas CTL, $f_1 \wedge f_2$ e $\neg f_1$ são fórmulas CTL.
3. Se f_1 e f_2 são fórmulas CTL, $\mathbf{E}f_1\mathbf{U}f_2, \mathbf{A}f_1\mathbf{U}f_2, \mathbf{EX}f_1$ e $\mathbf{AX}f_1$ são fórmulas CTL.

Na Figura 2.2 apresenta-se, graficamente, seqüências de estados que satisfazem a algumas fórmulas CTL básicas.

2.3.3 Comparação semântica entre LTL e CTL

Existe uma diferença semântica grande entre as lógicas LTL e CTL. Uma fórmula LTL estabelece uma regra para uma determinada trajetória de um modelo, independentemente de outras trajetórias que possam ser definidas. Uma fórmula CTL, ao contrário, estabelece regras a serem obedecidas por um determinado conjunto de estados e leva em consideração todos os possíveis futuros que possam existir a partir destes estados. Dadas estas diferenças, não é de se estranhar que ambas as lógicas possuam diferentes expressividades, como pode ser exemplificado abaixo:

- A fórmula $\mathbf{EX}p$ não pode ser traduzida para a lógica LTL, já que nenhuma fórmula LTL pode expressar a possibilidade de uma ramificação a partir de um estado de um trajetória.

- A fórmula $\mathbf{GF}p_1 \rightarrow \mathbf{GF}p_2$ não pode ser traduzida para a lógica CTL. A fórmula indica que p_2 deve ocorrer um número infinito de vezes no futuro sempre que o mesmo ocorrer com p_1. A afirmação não tem como ser representada em CTL porque a semântica deste tipo de lógica temporal não consegue representar o comportamento de uma trajetória à medida que ela evolui.

2.4 Verificação de modelos

O objetivo da verificação de modelos é determinar se um dado modelo satisfaz a uma dada propriedade. Vários diferentes algoritmos têm sido usados com sucesso para esta tarefa, usando diferentes lógicas temporais e estruturas de dados. Uma vez que a propriedade não satisfeita é determinada, um verificador de modelos pode retornar um exemplo de como esta violação ocorre. Esta violação é apresentada por um contra-exemplo.

Contra-exemplos de propriedades escritas em LTL são definidas usando seqüências lineares. Em contraste, as propriedades em CTL, são fórmulas de estado. Portanto, o problema da verificação de modelos a partir de fórmulas CTL é encontrar um conjunto de estados que satisfaz uma dada fórmula em uma dada estrutura de Kripke (21).

A primeira abordagem aplicada com sucesso na verificação de modelos é a verificação de modelos explícita. Existem diferentes abordagens baseadas em propriedades escritas em LTL e CTL ((22), (23)). Em todas as abordagens, o espaço de estado é representado explicitamente, e é identificado por meio da exploração progressiva, uma violação da propriedade que está sendo verificada. Por exemplo, em uma verificação de modelos de uma propriedade LTL, a negação de uma propriedade é representada como um autômato que aceita palavras infinitas. Se o produto síncrono de um modelo e um autômato contém qualquer caminho aceitável, então este caminho prova a violação da propriedade (o caminho mostra que a negação da propriedade é aceita pelo modelo do autômato). Os algoritmos de exploração podem tanto ser de profundidade ou largura; recentemente a exploração heurística também tem sido considerada (24). A exploração por largura sempre encontra o contra-exemplo com o caminho mais curto possível, mas a demanda de memória é significativamente maior do que a exploração por profundidade.

Para a exploração dos estados no modelo e detecção de violações nas propriedades, foram desenvolvidos verificadores de modelos, como será apresentação na Seção 2.5.

2.5 Verificador de modelo - NuSMV

A ferramenta adotada para este trabalho para a verificação de modelos foi o NuSMV (25), principalmente pelos seguintes motivos:

- é uma ferramenta código livre e pode ser gratuitamente adquirida através da URL *http://nusmv.irst.itc.it.*
- possui uma boa documentação que auxilia na construção dos modelos e das especificações em LTL e CTL;
- permite trabalhar com arquivos de entrada contendo a descrição do modelo;

O NuSMV é uma re-implementação e extensão do SMV (*Symbolic Model Verifier* (26), desenvolvida na Universidade Carnegie Mellon. O objetivo principal do NuSMV é verificar se um modelo satisfaz um conjunto desejado de propriedades especificadas pelo usuário. No NuSMV, as especificações a serem verificadas podem ser expressas em duas diferentes lógicas temporais: A CTL (*Computing Tree Logic*) e a LTL (*Linear Temporal Logic*). Quando uma propriedade não é satisfeita pelo modelo, o NuSMV apresenta um contra-exemplo, formado por um traço da máquina de estados finitos que levou a violação da propriedade.

A descrição do modelo no NuSMV é baseada na definição das variáveis que compõem o modelo, de seus valores iniciais e dos valores que estas variáveis assumem após uma transição de estados.

Para este trabalho, foram utilizadas as seguintes declarações do NuSMV para a construção do modelo: `MODULE`, `VAR`, `INIT` e `TRANS` (27).

Para exemplificar a utilização do NuSMV, consideremos o modelo apresentado na Figura 2.3.

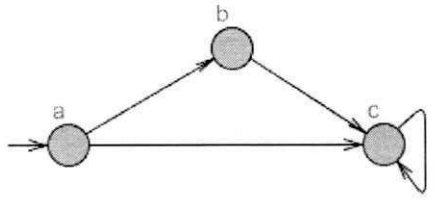

Figura 2.3: Modelo formado pelo diagrama de transição de estados

Na Figura 2.4, é apresentado na semântica do NuSMV, o modelo apresentado na Figura 2.3 e a especificação que deseja-se verificar. A declaração `MODULE` é utilizada para encapsular outras declarações, de forma que este modelo poderá ser utilizado como um módulo em outras aplicações. O identificador imediatamente seguinte a declaração

`MODULE` é o nome associado ao módulo. A declaração `VAR` é utilizada para criar as variáveis utilizadas no modelo em *NuSMV*. A declaração `INIT` é utilizada para gravar o valor inicial das variáveis declaradas. Os diferentes valores iniciais declarados no modelo gerado, representam os possíveis estados iniciais do modelo. A declaração `TRANS` define uma relação de transição de estados das variáveis e é utilizada para representar a máquina de estados finitos que forma o modelo. A declaração `SPEC` é utilizada para inserir as especificações em LTL ou CTL que serão verificadas. A especificação que será testada afirmar que *sempre globalmente* "AG" que o estado da variável é "estado = a" o *próximo estado da variável sempre será* "AX" "estado = c".

```
MODULE main
  VAR
    estado:{a, b, c};
  INIT
    (estado = a)
  TRANS
    (estado = a & next(estado) = b)|
    (estado = b & next(estado) = c)|
    (estado = a & next(estado) = c)|
    (estado = c & next(estado) = c)

  SPEC AG ((estado = a) -> AX(estado = c))
```

Figura 2.4: Modelo escrito na semântica do NuSMV

Na Figura 2.5, é apresentado o resultado da verificação do modelo pelo NuSMV. Nesta verificação, um contra-exemplo foi gerado comprovando uma violação da especificação, onde o estado da variável "estado = b" foi encontrado depois do "estado=a".

```
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification AG (estado = a -> AX estado = c)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  estado = a
-> Input: 1.2 <-
-> State: 1.2 <-
  estado = b
```

Figura 2.5: Contra-exemplo encontrado pelo NuSMV

Capítulo 3

Método Proposto

Neste capítulo, introduz-se o método para a verificação formal automática de programas a partir do seu modelo reduzido gerado com informações dos traços das execuções. Este método tem por objetivo auxiliar os programadores na detecção de erros introduzidos durante a implementação do código fonte do programa, como já foi discutido no Capítulo 1. Assume-se que o código em questão não foi submetido a nenhuma verificação formal.

Como ilustrado na Figura 3.1, a primeira etapa do método consiste na instrumentação do código fonte do programa que será verificado, com a finalidade de coletar e armazenar a transição de estados das variáveis durante execuções do programa em um arquivo de histórico de execução, doravante denominado *log*. Este conjunto de transições em uma execução é denominada traços ou caminhos, doravante denominado apenas de traços. A segunda etapa do processo consiste na execução do código fonte instrumentado, para a coleta dos traços e criação do arquivo de *log*. Após a criação do arquivo de *log*, a terceira etapa do processo é iniciada, onde aplica-se o algoritmo para o tratamento dos traços e a geração de um modelo reduzido do programa. Na quarta etapa, o modelo reduzido obtido é formatado para a semântica de alguma ferramenta de verificação formal (ex, SMV, NuSMV, Spin, Uppaal, etc). Na quinta etapa, o verificador de modelos é executado para confrontar as especificações e o modelo reduzido formatado. Caso a especificação não seja satisfeita, o verificador de modelos apresentará um contra-exemplo, ou seja, uma seqüência de transição de estados (traços) que levou a falha. Por último, a falha deve ser analisada e rastreada no código fonte, o que auxiliará na correção do código. O processo deve ser reiniciado para a verificação de novas falhas.

Nas seções seguintes as etapas do método introduzido são discutidas.

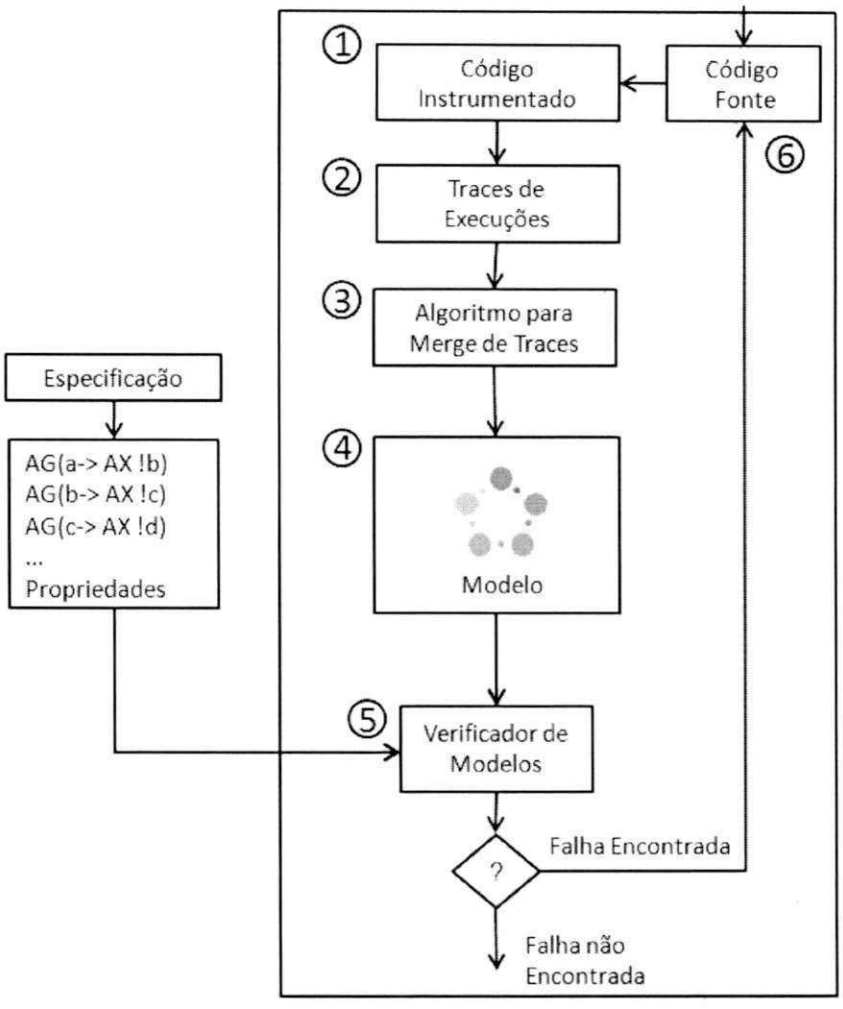

Figura 3.1: Diagrama ilustrando as etapas para a execução do método proposto

3.1 Instrumentação do código

O processo de instrumentação de código consiste na de adição de código fonte com funções secundárias ao programa, ou seja, são de fato anotações ao código do produto. No método introduzido, a instrumentação tem por finalidade coletar e armazenar as transições de estado das variáveis que serão utilizadas para computar o modelo reduzido do programa.

A implementação da instrumentação depende da linguagem de programação utilizada, ou seja, dos métodos e técnicas de programação empregadas para a coleta e armazenamento das informações. No contexto deste trabalho, como se evidenciará no estudo de caso apresentado no Capítulo 4, as transições de estado das variáveis são capturadas por meio da chamada de um método que armazena localmente os estados das variáveis e ao completar a execução do programa, um arquivo de *log* é criado, para o armazenamento

permanente das informações.

3.2 Execução do programa para geração dos traços do código

Após a etapa de instrumentação do código, o programa deve ser compilado e executado para que os traços sejam gerados e capturados no arquivo de *log*.

Na Figura 3.2, apresentam-se exemplos de traços da execução de uma programa. As variáveis X e Y são selecionadas, durante a instrumentação do código, para a formação do modelo. Os possíveis valores detectados durante as execuções são: X1 e X2, para a variável X, e, Y1 e Y2 para a variável Y.

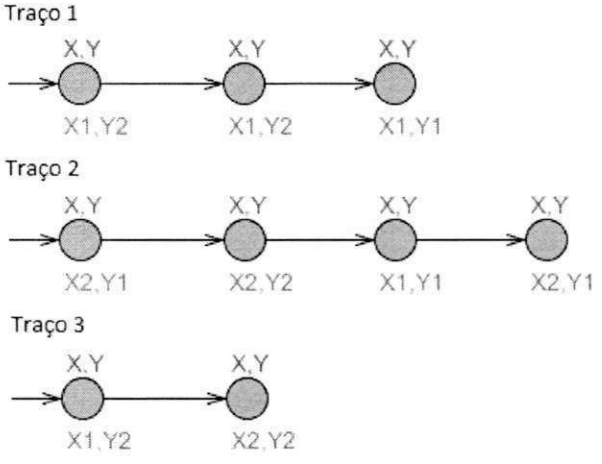

Figura 3.2: Estados das variáveis X e Y capturados durante execuções do programa

O comprimento de cada traço é determinado pela quantidade de transições de estados, tendo o tamanho 3 para o traço 1, 4 para o traço 2 e 2 para o traço 3. O comprimento de cada traço depende da lógica implementada no programa e uma quantidade maior de informação desse programa está presente em um traço de tamanho maior.

A descoberta da quantidade ideal de traços que deve ser gerada para a obtenção de um modelo ideal não é conhecida e não faz parte do escopo deste trabalho. Uma abordagem que pode ser utilizada para uma melhor qualidade dos traces é a verificação da negação da especificação antes da verificação da especificação na Etapa 5 da Figura 3.1. O verificador de modelos deve apresentar um contra-exemplo para cada negação da especificação, o que garante que o modelo gerado pelos traços acessou pelo menos uma parte do código onde a especificação foi implementada. Caso nenhum contra-exemplo tenha sido encontrado, um maior número de traços deve ser gerado para que o modelo contemple uma parte do

código onde a especificação é implementada.

3.3 Algoritmo para merge dos traços para a geração do modelo reduzido do programa

Nesta etapa, um algoritmo é aplicado nos traços coletados para a construção de um modelo reduzido do programa. O algoritmo implementado é obtido a partir do algoritmo *Timed-Preserving Merge* (20), sendo este alterado para manipulação de múltiplas variáveis e traços de comprimentos diferentes, conforme apresentado no Algoritmo 1 e doravante denominado Algoritmo Merge.

Neste algoritmo, define-se o conceito de *estado auxiliar*. Dado que s seja um estado e $i \in \mathbb{N}$, com $i \geqslant 1$. Então o par (s, i) é um *estado auxiliar*. Além disso, denomina-se que i é um instante. O *estado auxiliar* será utilizado durante o algoritmo, pois existe a necessidade de discretização no tempo do instante de cada estado no trace. Assim, cada estado será convertido em um *estado auxiliar*, formado pelo estado no trace e o correspondente instante no traço.

No Algoritmo 1, a entrada é formada pelo conjunto de possíveis estados das variáveis e um conjunto de traços de estados. A saída desejada é um espaço de estados que forma o modelo reduzido do programa.

Nas linhas de **1-3**, define-se os conjuntos que armazenam as informações para a formação da máquina de estados finita $<S', R, I>$. Sendo S', o conjunto formado pelos estados auxiliares da cada estado dos traços, R, o conjunto de transições de estados auxiliares, e I, o conjunto de estados auxiliares iniciais. Na linha **4**, ocorre a varredura dos traços para a coleta do estado inicial de cada traço, que será convertido para um estado auxiliar com instante $i = 1$. Na linha **6**, o estado auxiliar inicial é armazenado no conjunto S'. Na linha **7**, o estado auxiliar inicial é armazenado no conjunto I.

Na linha **9**, ocorre uma nova varredura dos traços para a formação do conjunto de transição de estados auxiliares R, e a continuação da formação do conjunto S'. Na linha **10**, define-se o comprimento de cada traço, sendo esse valor utilizando para a varredura dos estados no traço. Na linha **11**, ocorre a varredura dos estados no traço. Na linha **12** e **13**, define-se o estado atual e o consecutivo. Nas linhas **14** e **15**, criam-se o estado atual auxiliar e o estado consecutivo auxiliar. Na linha **16**, o estado auxiliar atual é inserido no conjunto S'. Na linha **17**, a transição do estado auxiliar atual para o estado auxiliar consecutivo é inserida no conjunto R.

O espaço de estados formado por S', R e I é completado após a varredura de todos os estados em todos os traços.

Este algoritmo possibilita aplicar todas as informações disponíveis nos traços na cons-

trução do modelo, ao contrário dos outros algoritmos apresentados no trabalho de Salem et al. (20). Além disso, para esse algoritmo, assume-se que se uma mesma transição de estado do programa ocorre em uma mesma seqüência, em dois traços diferentes, então, as futuras transições de um traço, a partir desse ponto, podem ser seguidas pelas transições de estado do outro traço e vice-versa permitindo um modelo reduzido mais representativo com um número menor de traces.

Entrada: Um conjunto S de possíveis estados e um conjunto T de traços de estados de S, onde cada estado é formado pelas variáveis $x, y, z, ..., n$

Saída: Um espaço de estados

1 *Dado que S' seja um conjunto vazio*;
2 *Dado que R seja uma relação binária vazia*;
3 *Dado que I seja um conjunto vazio*;
4 **foreach** $t \in T$ **do**
5 | *Dado que S_1 $(x_1, y_1, ..., n_1)$ seja o estado inicial de t*;
6 | $S' \leftarrow S'\ U\ \{(x_1, y_1, ..., n_1, 1)\}$;
7 | $I \leftarrow I\ U\ \{(x_1, y_1, ..., n_1, 1)\}$;
8 **end**
9 **foreach** $t \in T$ **do**
10 | *Dado que n seja o comprimento do traço t em T*;
11 | **for** $i \leftarrow 1$ **to** $n - 1$ **do**
12 | | *Dado que s_i seja o $i - esimo$ estado em t*;
13 | | *Dado que s_{i+1} seja o $(i + 1) - esimo$ estado em t*;
14 | | *Dado que eA_1 seja um estado auxiliar formado pela dupla(s_i, i)*;
15 | | *Dado que eA_2 seja um estado auxiliar formado pela dupla$(s_{i+1}, i + 1)$*;
16 | | $S' \leftarrow S'\ U\ \{eA_2\}$;
17 | | $R \leftarrow R\ U\ \{(eA_1, eA_2)\}$;
18 | **end**
19 **end**

Resultado: S', R, I

Algoritmo 1: Algoritmo Merge

Como exemplo, o Algoritmo 1 será aplicado nos traços apresentados na Figura 3.2.

A partir das informações contidas nos traços pode-se retirar as entradas S e T do Algoritmo Merge, conforme as expressões abaixo:

$$S = \{(\mathsf{X1},\mathsf{Y1}), (\mathsf{X1},\mathsf{Y2}), (\mathsf{X2},\mathsf{Y1}), (\mathsf{X2},\mathsf{Y2})\}$$

$$T = \{[(\mathsf{X1}, \mathsf{Y2}), (\mathsf{X1}, \mathsf{Y2}), (\mathsf{X1}, \mathsf{Y1})]; [(\mathsf{X2}, \mathsf{Y1}), (\mathsf{X2}, \mathsf{Y2}), (\mathsf{X1}, \mathsf{Y1}), (\mathsf{X2}, \mathsf{Y1})]; [(\mathsf{X1}, \mathsf{Y2}), (\mathsf{X2}, \mathsf{Y2})]\}$$

Após aplicação do algoritmo, as saídas *I*, *R* e *S'* são obtidas conforme expressões abaixo.

$$I = \{(\mathsf{X1}, \mathsf{Y2}, 1), (\mathsf{X2}, \mathsf{Y1}, 1)\}$$

$$\begin{aligned} R = \{ \quad & [(\mathsf{X1}, \mathsf{Y2}, 1), (\mathsf{X1}, \mathsf{Y2}, 2)], [(\mathsf{X1}, \mathsf{Y2}, 2), (\mathsf{X1}, \mathsf{Y1}, 3)], [(\mathsf{X2}, \mathsf{Y1}, 1), (\mathsf{X2}, \mathsf{Y2}, 2)], \\ & [(\mathsf{X2}, \mathsf{Y2}, 2), (\mathsf{X1}, \mathsf{Y1}, 3)], [(\mathsf{X1}, \mathsf{Y1}, 3), (\mathsf{X2}, \mathsf{Y1}, 4)], [(\mathsf{X1}, \mathsf{Y2}, 1), (\mathsf{X2}, \mathsf{Y2}, 2)]\} \end{aligned}$$

$$S' = \{ \quad (\mathsf{X1}, \mathsf{Y2}, 1), (\mathsf{X1}, \mathsf{Y2}, 2), (\mathsf{X1}, \mathsf{Y1}, 3), (\mathsf{X2}, \mathsf{Y1}, 1), (\mathsf{X2}, \mathsf{Y1}, 4), (\mathsf{X2}, \mathsf{Y2}, 2)\}$$

Com as informações de *I*, *R* e *S'*, o modelo reduzido do programa é representado na Figura 4.3.

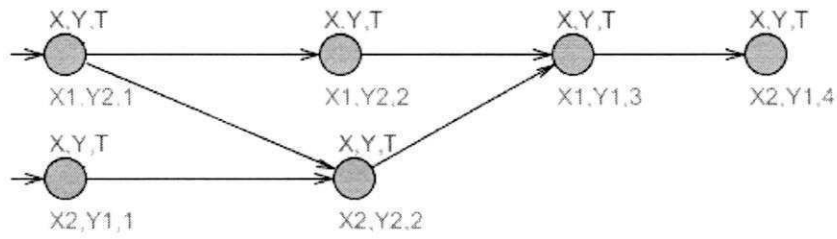

Figura 3.3: Modelo reduzido do programa gerado após a aplicação do algoritmo

3.4 Verificação do modelo e correção de erros

Nesta etapa é realizada a verificação do modelo gerado utilizando uma ferramenta de verificação de modelos, conforme ilustrado na Figura 3.4. O verificador de modelos confronta a especificação do programa com o modelo gerado. Caso alguma falha seja detectada, o verificador de modelos apresentará um traço com as transições de estados das variáveis que levaram à falha.

A partir da informação do traço que leva a falha, é possível ao programador interpretar e rastrear no código fonte os trechos de código incorretos. Após a correção do código, um novo modelo deve ser gerado e uma nova verificação do modelo aplicada, para confirmar a correção do erro no programa.

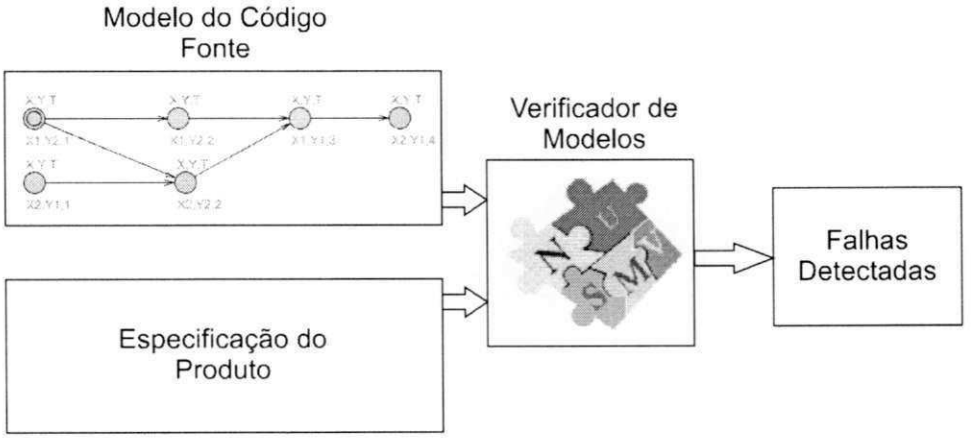

Figura 3.4: Esquema para a verificação do modelo gerado

Capítulo 4

Estudo de Caso

Neste capítulo um sistema para limpeza do pára-brisa veicular é apresentado como estudo de caso. Este sistema foi escolhido devido a sua recorrente utilização na área de verificação formal (28, 29) e pela facilidade de especificá-lo por ser um sistema bastante conhecido e utilizado.

Inicialmente o sistema será definido, especificado e implementado para que as etapas do método introduzido no Capítulo 3 sejam aplicadas.

Neste capítulo, apenas trechos de código são apresentados, para o código completo acessar: http://dl.dropbox.com/u/6426894/cfonte.zip.

4.1 Definição do sistema para limpeza do pára-brisa

Nos automóveis, o sistema para limpeza do pára-brisa é necessário para manter a visibilidade dos motoristas. Este sistema é formado por um módulo controlador, por motores das paletas limpadoras, a bomba d'água dos bicos ejetores e os controles. As paletas limpadoras, quando utilizadas isoladamente, possuem a finalidade de retirar a água do pára-brisa, necessárias principalmente durante as chuvas. Estas paletas também podem ser utilizadas em conjunto com os bicos ejetores de água. Neste caso, os bicos ejetores lançam jatos de água no pára-brisa, que em seguida serão removidos pelas paletas limpadoras. Este procedimento geralmente é utilizado para a retirada de sujeira acumulada no pára-brisa. O módulo controlador é programado para ativar os motores e a bomba d'água de acordo com o estado dos controles. Na Figura 4.1 é ilustrado um sistema para limpeza de um pára-brisa veicular.

Os sistemas tradicionais são comandados pelo motorista através de um controle manual, geralmente posicionado próximo ao volante. No estudo de caso apresentado neste capítulo, o controle permite que o motorista selecione o estado desligado, velocidade baixa ou velocidade alta para os motores das paletas limpadoras. O controle também permite

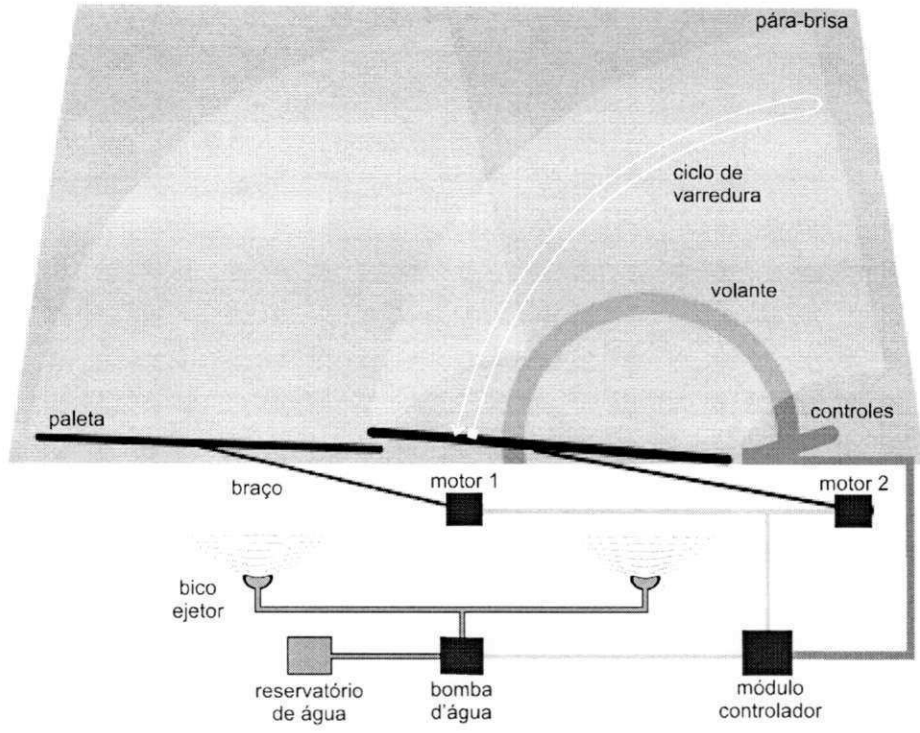

Figura 4.1: Sistema para limpeza de um pára-brisa veicular

que os estados ativado ou desativado da bomba d'água dos bicos ejetores sejam selecionados.

Caso os motores estejam operando na velocidade baixa ou alta, quando a bomba d'água é ativada, os motores devem continuar o movimento de varredura com a mesma velocidade selecionada, durante e após a ativação da bomba d'água. Caso os motores estejam desligados, enquanto o motorista ativa a bomba d'água, as paletas devem automaticamente iniciar três ciclos de varredura do pára-brisa em velocidade baixa, assim que a bomba d'água for desativada.

O movimento de varredura das paletas é realizado através da força dos motores (motor 1, motor 2) transferida aos braços conectados nas paletas. A água, lançada pelos bicos ejetores, é bombeada através da bomba d'água, que coleta água do reservatório e impõe pressão nos bicos ejetores.

O módulo controlador é o componente programável do sistema, ativa as saídas (motor 1, motor 2 e bomba d'água) de acordo com a lógica de programação implementada e o estado do controle dos motores das paletas limpadoras e da bomba d'água dos bicos ejetores.

Sendo o programa implementado no módulo controlador suscetível a erros durante o processo de desenvolvimento, o método proposto será aplicado no programa implementado como uma alternativa adicional de verificação para garantir o correto funcionamento do

sistema.

4.1.1 Requisitos da especificação do sistema para a limpeza do pára-brisa

O módulo controlador é o componente programável do sistema de limpeza do pára-brisa e, portanto, sua correta implementação é necessária para que as operações de limpeza sejam executadas. O programa do módulo controlador é implementado a partir do conjunto de requisitos da especificação do sistema que devem ser seguidas pelo programador.

A partir da descrição do sistema alvo introduzida na Seção 4.1, os requisitos da especificação do módulo controlador são apresentados abaixo.

1. As entradas do módulo controlador são formadas pelo estado do controle das paletas e pelo controle dos bicos ejetores.
   (a) Os estados do controle das paletas são: *Desligado*, *Velocidade Baixa* ou *Velocidade Alta*;
   (b) Os estados do controle dos bicos ejetores são: *Ativado* ou *Desativado*;
2. As saídas do módulo controlador são os estados do motor 1, do motor 2 e a bomba d'água.
   (a) Os estados dos motores 1 e motor 2 são: *Desligado*, *Velocidade Baixa* ou *Velocidade Alta*;
   (b) Os estados da bomba d'água são: *Ativado* ou *Desativado*;
3. Sempre que o estado do controle da paleta for *Velocidade Baixa*, o estado do motor 1 e motor 2 também deve ser *Velocidade Baixa*.
4. Sempre que o estado do controle da paleta for *Velocidade Alta*, o estado do motor 1 e motor 2 também deve ser *Velocidade Alta*.
5. Sempre que o estado do controle dos bicos ejetores for *Desativado*, o estado da bomba d'água também deve ser *Desativado*.
6. Sempre que o estado do controle dos bicos ejetores for *Ativado*, o estado da bomba d'água também deve ser *Ativado*.
7. Sempre após a transição do estado do controle dos bicos ejetores de *Ativado* para *Desativado*, o estado dos motores 1 e 2 deve ser:

(a) Caso o estado do controle das paletas for *Desligado*, o estado dos motores 1 e motor 2 deve ser *Velocidade Baixa* durante o período de 3 ciclos de varredura e em seguida *Desligado*.

(b) Caso o estado do controle das paletas seja *Velocidade Baixa* ou *Velocidade Alta*, os requisitos 3 ou 4 devem ser seguidos.

A partir destes requisitos, define-se os estados do módulo controlador em: Ocioso, Ejetando Água, Esperando e Limpando, conforme a Figura 4.2.

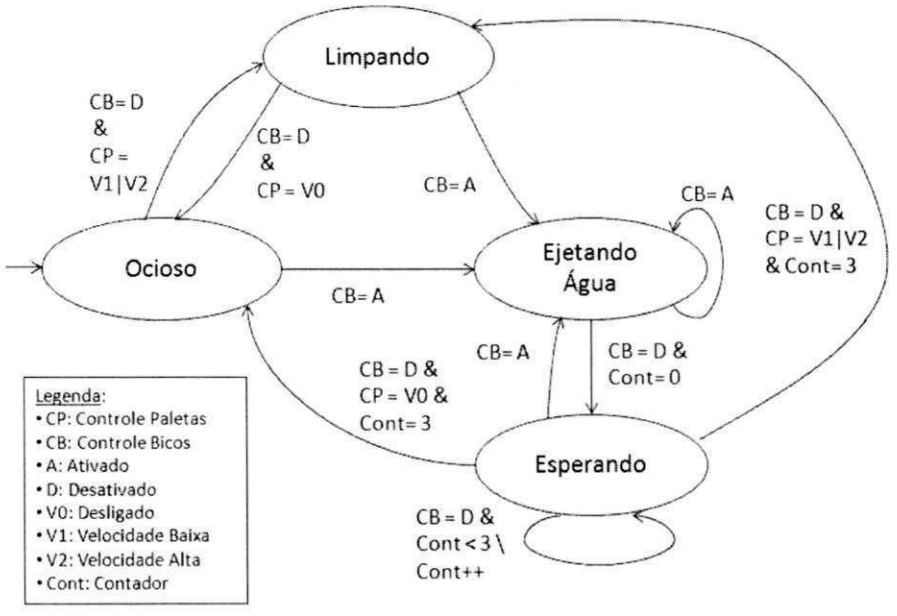

Figura 4.2: Estados do Módulo Controlador

4.1.2 Programação do sistema para a limpeza do pára-brisa

Nas aplicações industriais, o módulo controlador geralmente é um módulo eletrônico com um microcontrolador programável em linguagem *C*. Neste trabalho, o módulo controlador é implementado em nível de simulação em computador, sendo a linguagem de programação Java escolhida devido as facilidades didáticas e de implementação.

Para a implementação da simulação do sistema de limpeza do pára-brisa, foram criados os seguintes componentes Controlador, Principal, Entradas e Saídas, conforme representação na Figura 4.3. O componente Entradas gera aleatoriamente o estado do controle das paletas e dos bicos ejetores. O componente Controlador captura os valores gerados pelo componente Entradas e processa estes valores através de uma lógica de decisão programada que decide o estado das variáveis do componente Saídas. O componente Saídas armazena o estado dos motores e da bomba d'água que é atualizado pelo componente Controlador.

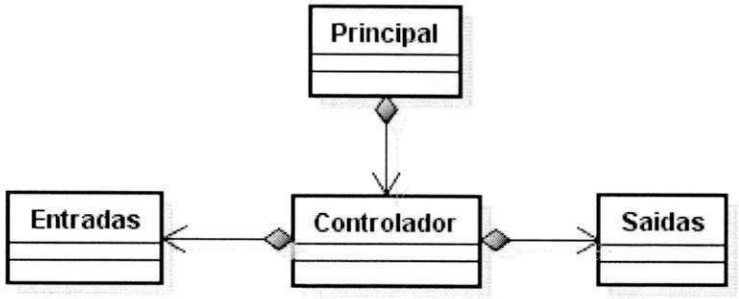

Figura 4.3: Diagrama de classes do programa

O componente Principal, implementado através da classe **Principal** é utilizado para iniciar a simulação e iniciar o componente Controlador, conforme o trecho do código apresentado na Listagem 4.1.

Listagem 4.1: Classe Principal

```
public class Principal {

        public static void main(String[] args) {

                Controlador moduloControlador = new Controlador();

        moduloControlador.iniciar();

    }
}
```

O componente Entrada é implementado através da classe **Entradas**, conforme trecho do código apresentado na Listagem 4.2. Nessa classe, são definidas as variáveis controlePaletas e controleBicos, que representam respectivamente o estado do controle das paletas e dos bicos ejetores.

A variável controlePaletas é do tipo **int**, tendo os possível valores inteiros *0*, *1* ou *2*, gerados aleatoriamente, a partir da operação (geraNumeroAlet.nextInt(3))%3. Os possíveis valores *0*, *1* ou *2* da variável controlePaletas, representam respectivamente os estados *Desligado*, *Velocidade Baixa* ou *Velocidade Alta* do controle das paletas.

A variável controleBicos é do tipo **boolean**, tendo os possível valores *true* ou *false*, gerados aleatoriamente, a partir da operação geraNumeroAlet.nextBoolean(). Os possíveis valores *true* ou *false* da variável controleBicos, representam respectivamente os estados *Ativado* ou *Desativado* do controle dos bicos ejetores.

Listagem 4.2: Classe Entradas

```
int controlePaletas;
```

```
boolean controleBicos;

Random geraNumeroAlet = new Random();

controlePaletas = (geraNumeroAlet.nextInt(3)) % 3;
controleBicos = geraNumeroAlet.nextBoolean();
```

O componente Controlador é implementado através da classe **Controlador**, conforme trecho do código apresentado na Listagem 4.3. Nesta classe, define-se a variável estadoModuloControlador do tipo **String** que armazenará os valores *"EjetandoAgua"*, *"Ocioso"*, *"Esperando"* ou *"Limpando"*, que caracterizam o estado do módulo controlador e que são determinados de acordo com o estado das variáveis de entrada controlePaletas e controleBicos, conforme foi apresentado na Figura 4.2.

Listagem 4.3: Classe Controlador

```
int controlePaletas;
boolean controleBicos;
int contador = 0;
String estadoModuloControlador;
Saidas saidasMC = new Saidas();
...

if ((controleBicos == false) & (controlePaletas == 0)
        & (estadoModuloControlador != "EjetandoAgua")) {
        saidasMC.setEstadoBombaDagua(false);
        saidasMC.setEstadoMotores(0);
        estadoModuloControlador = "Ocioso";
}
if ((controlePaletas > 0) & (controleBicos == false)
                & (estadoModuloControlador != "Esperando")) {
        estadoModuloControlador = "Limpando";
        saidasMC.setEstadoBombaDagua(false);
        saidasMC.setEstadoMotores(controlePaletas);
}
if (controleBicos == true) {
        saidasMC.setEstadoBombaDagua(true);
        saidasMC.setEstadoMotores(controlePaletas);
        estadoModuloControlador = "EjetandoAgua";
}
if ((estadoModuloControlador == "EjetandoAgua")
                & (controleBicos == false)) {
        if (controlePaletas == 0) {
                saidasMC.setEstadoMotores(1);
                contador = 0;
```

```
30          } else {
                    saidasMC.setEstadoMotores(controlePaletas);
                    contador = 0;
            }
            estadoModuloControlador = "Esperando";
35          saidasMC.setEstadoBombaDagua(false);
    }
    if (estadoModuloControlador == "Esperando") {
            if ((contador > 2) & (controlePaletas > 0)) {
                    estadoModuloControlador = "Limpando";
40                  saidasMC.setEstadoMotores(controlePaletas);
                    saidasMC.setEstadoBombaDagua(false);
                    contador = 0;
            } else if ((contador > 2) & (controlePaletas == 0)) {
                    estadoModuloControlador = "Ocioso";
45                  saidasMC.setEstadoMotores(controlePaletas);
                    saidasMC.setEstadoBombaDagua(false);
                    contador = 0;
            } else {
                    contador++;
50          }
    }
```

O estado das variáveis do componente Saídas é atualizado de acordo com o resultado das operações com as variáveis de entrada do componente Entradas e do valor da variável estadoModuloControlador. O componente Saídas, é implementado através da classe **Saidas**, conforme trecho de código apresentado na listagem 4.4. Nesta classe, são definidas as variáveis estadoMotores e estadoBombaDagua, que respectivamente, guardam o estado dos motores 1 e 2 e da bomba d'água.

Listagem 4.4: Classe Saidas

```
int estadoMotores = 0;
boolean estadoBombaDagua;

public int getEstadoMotores() {
5           return estadoMotores;
    }
    public void setEstadoMotores(int estadoMotores) {
            this.estadoMotores = estadoMotores;
    }
10  public boolean isEstadoBombaDagua() {
            return estadoBombaDagua;
    }
    public void setEstadoBombaDagua(boolean estadoBombaDagua) {
```

```
        this.estadoBombaDagua = estadoBombaDagua;
15  }
```

A variável estadoMotores é do tipo **int**, sendo *0*, *1* ou *2* os possíveis valores, que respectivamente representam os estados *Desligado*, *Velocidade Baixa* e *Velocidade Alta* dos motores 1 e 2. Estes estados são gravados pelos componente Controlador através do método setEstadoMotores.

A variável estadoBombaDagua é do tipo **boolean**, sendo *true* ou *false* os possíveis valores que respectivamente representam os estados *Ativado* ou *Desativado* da bomba d'água. Estes estados são gravados pelo componente Controlador através do método setEstadoBombaDagua.

4.2 Aplicação das etapas do método proposto

A partir do programa implementado, inicia-se a aplicação das etapas do método introduzido neste trabalho. Como foi apresentado no Capítulo 3, as etapas são formadas pela instrumentação do código, execução do programa para a captura dos traços, aplicação do algoritmo para a criação do modelo reduzido a partir dos traços, verificação do modelo e correção dos erros no código.

4.2.1 Instrumentação do código para geração dos traces

Nesta seção, a etapa da instrumentação do código é aplicada ao estudo de caso. As classes **Principal** e **Controlador** apresentadas na Seção 4.1.2, são as únicas classes afetadas na etapa de instrumentação. E a classe **Log**, foi criada para tratar o armazenamento dos traços em uma arquivo de *log*.

Com relação a classe **Controlador**, a implementação da instrumentação consistiu na inclusão do método salvaEstado() nos locais onde o estado das variáveis devem ser capturados, para a construção do modelo reduzido, conforme trecho de código apresentado na listagem 4.5. Cada vez que o método salvaEstado() é invocado, o estado das variáveis que formam o modelo reduzido do programa é salvo em uma variável do tipo **LinkedList** que no final da execução armazenará o traço da execução do programa.

Listagem 4.5: Classe Controlador Instrumentada

```
    traceMC = new LinkedList<String>();
    traceCP = new LinkedList<String>();
    traceCB = new LinkedList<String>();
    traceEM = new LinkedList<String>();
5   traceEB = new LinkedList<String>();
```

```
for (int i = 0; i < numeroDeOperacoes; i++) {

        controlePaletas = estadoControles.getControlePaletas();
        controleBicos = estadoControles.isControleBicos();
        estadoControles.gerarEntradasAlet();

    if ((controleBicos == false) & (controlePaletas == 0)
                    & (estadoModuloControlador != "EjetandoAgua")) {
            saidasMC.setEstadoBombaDagua(false);
            saidasMC.setEstadoMotores(0);
            estadoModuloControlador = "Ocioso";
            salvaEstado();
        }

        if ((controlePaletas > 0) & (controleBicos == false)
                    & (estadoModuloControlador != "Esperando")) {
            estadoModuloControlador = "Limpando";
            saidasMC.setEstadoBombaDagua(false);
            saidasMC.setEstadoMotores(controlePaletas);
            salvaEstado();
        }
    ...
}

public void salvaEstado() {
            traceMC.add(estadoModuloControlador);
            traceCP.add(String.valueOf(controlePaletas));
            traceCB.add(String.valueOf(controleBicos));
            traceEM.add(String.valueOf(saidasMC.estadoMotores));
            traceEB.add(String.valueOf(saidasMC.estadoBombaDagua));
}
```

Com relação a classe **Principal**, é necessário instrumentá-la para que as informações coletadas do programa sejam formatadas antes da escrita em um arquivo de *log*. Este procedimento tem por objetivo facilitar a manipulação dos dados quando realizada a aplicação do algoritmo, para a criação da representação do espaço de estados reduzido. A formatação é realizada conforme o trecho do código apresentado na Listagem 4.6.

Listagem 4.6: Classe Principal Instrumentada

```
public class Principal {

    public static void main(String[] args) {

        Controlador moduloControlador = new Controlador();
```

```
moduloControlador.iniciar();

File arquivoLog = new File("log.txt");
Log log = new Log(arquivoLog);

String nl = System.getProperty("line.separator");

// Captura os possíveis estados das variáveis
String infoLog = "EstadoMC" + nl;
for (String s : moduloControlador.getEstadosMC()) {
    infoLog += s + " ";     }
infoLog += nl + "ControlePaletas" + nl;
for (String s : moduloControlador.getEstadosCP()) {
    infoLog += s + " ";     }
infoLog += nl + "EstadoMotor" + nl;
for (String s : moduloControlador.getEstadosEM()) {
    infoLog += s + " ";     }
infoLog += nl + "ControleBicos" + nl;
for (String s : moduloControlador.getEstadosCB()) {
    infoLog += s + " ";     }
infoLog += nl + "EstadoBombaDagua" + nl;
for (String s : moduloControlador.getEstadosEB()) {
    infoLog += s + " ";     }

infoLog += nl + nl + "TRACES" + nl;

// Número de execuções que foram realizadas
int numeroExec = 3;

for (int k = 0; k < numeroExec; k++) {

        List<String> traceMC = moduloControlador.
            getListaDeTracesMC().get(k);
        List<String> traceCP = moduloControlador.
            getListaDeTracesCP().get(k);
        List<String> traceEM = moduloControlador.
            getListaDeTracesEM().get(k);
        List<String> traceCB = moduloControlador.
            getListaDeTracesCB().get(k);
        List<String> traceEB = moduloControlador.
            getListaDeTracesEB().get(k);

        ListIterator<String> iteratorMC = traceMC.
            listIterator();
        ListIterator<String> iteratorCP = traceCP.
```

```
                    listIterator();
40                  ListIterator<String> iteratorEM = traceEM.
                    listIterator();
                    ListIterator<String> iteratorCB = traceCB.
                    listIterator();
                    ListIterator<String> iteratorEB = traceEB.
                    listIterator();

                    while (iteratorMC.hasNext()) {
45
                        String estadoMC = iteratorMC.next();
                        String estadoCP = iteratorCP.next();
                        String estadoEM = iteratorEM.next();
                        String estadoCB = iteratorCB.next();
50                      String estadoEB = iteratorEB.next();

                        infoLog += estadoMC + " " + estadoCP + " "
                            + estadoEM + " " + estadoCB + " " +
                            estadoEB + " ";
                    }

55                  infoLog += nl;
                }

                log.salvarLog(infoLog);
            }
60  }
```

Para salvar os traços capturados em um arquivo de *log* a classe **Log** foi implementada, conforme o trecho do código apresentado na Listagem 4.7. O método salvarLog(String nomeArquivo) é utilizado pela classe **Principal** quando todas as informações dos traços estão prontas para serem armazenadas no arquivo de *log*.

Listagem 4.7: Classe Log

```
public class Log {

  private File arquivoLog;

5   public Log(File output){
      this.arquivoLog = output;
    }

    public void salvarLog(String nomeArquivo) throws IOException{
10    FileWriter gravar = new FileWriter(arquivoLog);
      BufferedWriter buf = new BufferedWriter(gravar);
```

```
        buf.write(nomeArquivo);
        buf.flush();
    }
}
```

Na Figura 4.4, é apresentado um exemplo de *log* salvo em uma arquivo no formato txt, gerado após cinco execuções do código fonte instrumentado, gerando conseqüentemente cinco traços. No início do arquivo de *log*, é apresentado os possíveis estados das variáveis coletadas durante a execução do programa. As seguintes denominações foram utilizadas para as variáveis no arquivo de *log*: EstadoMC(*Estado do Módulo Controlador*), ControlePaletas(*Estado do Controle das Paletas*), EstadoMotor(*Estado do Motor*), ControleBicos(*Estado do Controle dos Bicos Ejetores*) e EstadoBombaDadua(*Estado da Bomba D'água*). Em seguida, após a declaração da palavra TRACES, é apresentada a seqüencia de transição de estados das variáveis, ou seja, o traço da execução.

No exemplo do arquivo de *log*, apresentado na Figura 4.4, é destacado a transição de estados "Limpando 2 2 false false" para "Ocioso 0 0 false false" que representam respectivamente, a transição de estado do Módulo controlador de *Limpando* para *Ocioso*, do estado controle das paletas de *Velocidade Alta* para *Desligado*, do estado do Motor 1 e 2 de *Velocidade Alta* para *Desligado* e da permanência do estado do controle dos bicos ejetores e do estado da Bomba d'água em *Desligado*.

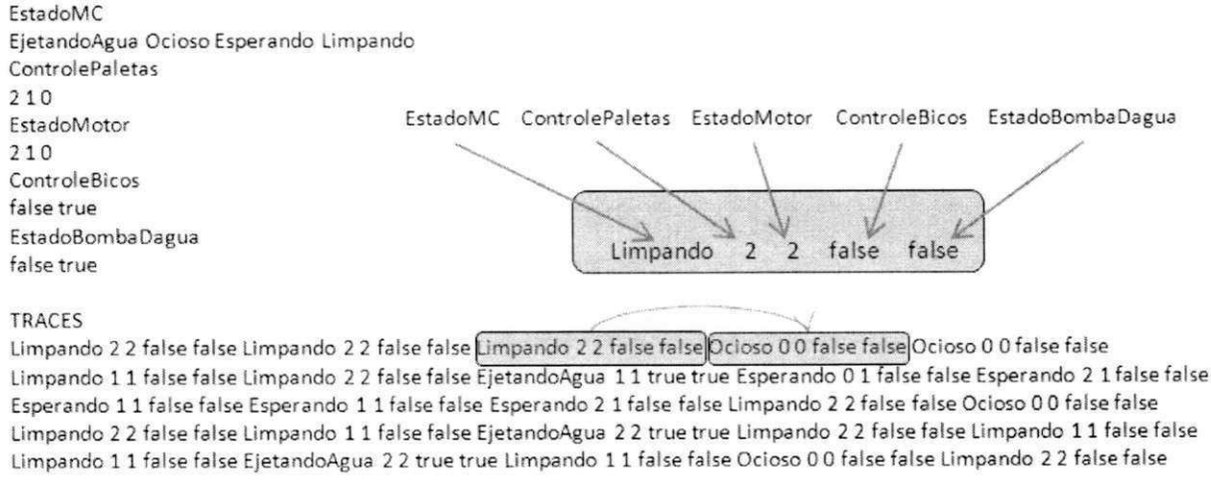

Figura 4.4: Exemplo do arquivo *log* gerado

Este arquivo de *log*, será utilizado para a criação da máquina de estados finitos que formará o modelo reduzido do programa, como apresentado na seção seguinte.

4.2.2 Aplicação do algoritmo para criação do espaço de estados reduzido

Nesta etapa será apresentada a implementação e aplicação do algoritmo para a criação da representação do espaço de estados finitos reduzido do programa com a semântica do NuSMV. Na Figura 4.5 são apresentadas as classes utilizadas na implementação.

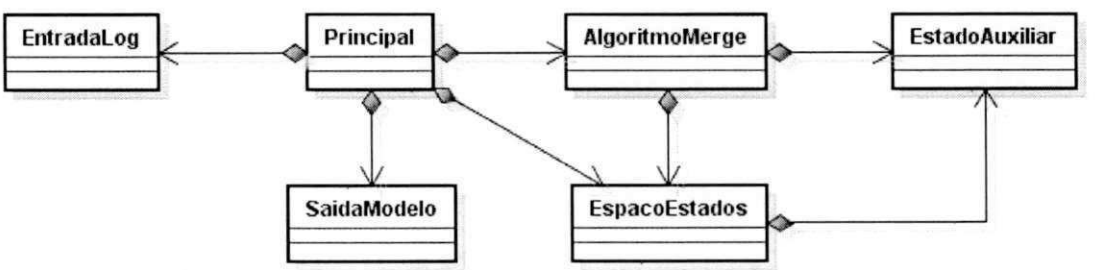

Figura 4.5: Principais classes do programa para criação do EEF reduzido

A classe **EntradaLog** é responsável por receber o arquivo de *log* com as informações coletadas durante as execuções do programa e, através da implementação do método parse, selecionar e armazenar os possíveis estados das variáveis e os traços das execuções armazenadas no *log*. Este procedimento, descrito conforme o trecho do código apresentado na Listagem 4.8, é necessário para o melhor tratamento das informações contidas no arquivo de *log*, durante a aplicação do algoritmo para a criação do modelo reduzido do programa.

Listagem 4.8: Classe EntradaLog

```
public class EntradaLog {

        private Set<String> estadosMC = new HashSet<String>();
        private Set<String> estadosCP = new HashSet<String>();
        private Set<String> estadosCB = new HashSet<String>();
        private Set<String> estadosEM = new HashSet<String>();
        private Set<String> estadosEB = new HashSet<String>();

        private List<List<String>> traces = new LinkedList<List<String>>();

        public EntradaLog(File input) throws FileNotFoundException,
            IOException {
                FileReader fReader = new FileReader(input);
                BufferedReader bReader = new BufferedReader(fReader);
                StreamTokenizer tokenizer = new StreamTokenizer(bReader);
```

```
            parse(tokenizer);
        }

20

        private void parse(StreamTokenizer tokenizer) throws IOException {

            // Parse up to the end of file
            for (int tok = tokenizer.nextToken(); tok !=
                StreamTokenizer.TT_EOF; tok = tokenizer
25                          .nextToken()) {

                if (tok == StreamTokenizer.TT_WORD) {
                    String word = tokenizer.sval;
                    if (word.equalsIgnoreCase("EstadoMC")) {
30                      parseNewLine(tokenizer);
                        parseEstadosMC(tokenizer);
                    } else if (word.equalsIgnoreCase("
                        ControlePaletas")) {
                        parseNewLine(tokenizer);
                        parseEstadosCP(tokenizer);
35                  } else if (word.equalsIgnoreCase("
                        ControleBicos")) {
                        parseNewLine(tokenizer);
                        parseEstadosCB(tokenizer);
                    } else if (word.equalsIgnoreCase("
                        EstadoMotor")) {
                        parseNewLine(tokenizer);
40                      parseEstadosEM(tokenizer);
                    } else if (word.equalsIgnoreCase("
                        EstadoBombaDagua")) {
                        parseNewLine(tokenizer);
                        parseEstadosEB(tokenizer);
                    } else if (word.equalsIgnoreCase("TRACES"))
                        {
45                      parseNewLine(tokenizer);
                        parseTraces(tokenizer);
                    }

                }

50

            }

        }
```

```
55 }
```

Na classe **AlgoritmoMerge**, utilizam-se as informações do *log* tratadas para a criação da representação do espaço de estados reduzido do programa.

Quando o método merge() é executado, a variável espacoEstados, do tipo **EspacoEstados**, é criada para armazenar as informações necessárias para a criação do modelo reduzido do programa. Inicialmente, o valor inicial das variáveis, contidas no arquivo *log*, EstadoMC, ControlePaletas, EstadoMotor, ControleBicos e EstadoBombaDagua, formarão os estados iniciais do modelo reduzido do programa. Em seguida, a transição de estados contida nos traços é discretizada no tempo e armazenada na variável espacoEstados, através do método adicionaTransicao(), conforme o trecho de código apresentado na listagem 4.9. Durante esse processo, as variáveis do tipo **EstadoAuxiliar**, são criadas para a manipulação e armazenamento dos estados obtidos de cada traço.

Listagem 4.9: Classe AlgoritmoMerge

```
public EspacoEstados merge() {

    EspacoEstados espacoEstados;

5       for (List<String> trace : traces) {
                String mc = trace.get(0);
                String cp = trace.get(1);
                String cb = trace.get(2);
                String em = trace.get(3);
10              String eb = trace.get(4);

                EstadoAuxiliar eS1 = new EstadoAuxiliar(mc, cp, cb, em, eb,
                    1);

                espacoEstados.adicionaEstadp(eS1);
15              espacoEstados.adicionaEstadoInicial(eS1);
        }

        for (List<String> trace : traces) {

20              int j = 1;
                for (int i = 1; i <= trace.size() - 5; i+=5) {
                        if (trace.size() > i) {

                                String mc = trace.get(i - 1);
25                              String cp = trace.get(i);
                                String cb = trace.get(i + 1);
                                String em = trace.get(i + 2);
```

```
                                String eb = trace.get(i + 3);

                                String mcI = trace.get(i + 4);
                                String cpI = trace.get(i + 5);
                                String cbI = trace.get(i + 6);
                                String emI = trace.get(i + 7);
                                String ebI = trace.get(i + 8);

                                EstadoAuxiliar x = new EstadoAuxiliar(mc,
                                    cp, cb, em, eb, j);
                                EstadoAuxiliar y = new EstadoAuxiliar(mcI,
                                    cpI, cbI, emI, ebI, j + 1);
                                j++;

                                espacoEstados.adicionaEstado(y);
                                espacoEstados.adicionaTransicao(x, y);
                            }
                        }
                    }
                return espacoEstados;
            }
```

A classe **EspacoEstados**, armazenará os estados iniciais e as transições do modelo reduzido do programa, a partir da execução dos métodos adicionaEstadoInicial e adicionaTransicao. Esses métodos são chamados durante a execução do objeto aplicaAlgorit, do tipo **AlgoritmoMerge**. A classe **EspacoEstados**, também implementa as funcionalidades para formatar as informações da representação do modelo reduzido do programa, gerando a representação no formato do verificador de modelos NuSMV, conforme (27). Esse processo é realizado através do método paraFormatoNuSMV(). Na listagem 4.10 é apresentado o trecho do código da classe **EspacoEstados**.

Listagem 4.10: Classe EspacoEstados

```
public class EspacoEstados {

    private Set<EstadoAuxiliar> estados = new HashSet<EstadoAuxiliar>()
        ;
    private Set<EstadoAuxiliar> estadosIniciais = new HashSet<
        EstadoAuxiliar>();

    private Set<String> estadosMC = new HashSet<String>();
    private Set<String> estadosCP = new HashSet<String>();
    private Set<String> estadosEM = new HashSet<String>();
    private Set<String> estadosCB = new HashSet<String>();
    private Set<String> estadosEB = new HashSet<String>();
```

```
    private Map<EstadoAuxiliar, Set<EstadoAuxiliar>> transicoes = new
        HashMap<EstadoAuxiliar, Set<EstadoAuxiliar>>();


    public EspacoEstados(Set<String> estadosMCi, Set<String> estadosCPi
        ,
                Set<String> estadosEMi, Set<String> estadosCBi,
                Set<String> estadosEBi) {

            this.estadosMC = estadosMCi;
            this.estadosCP = estadosCPi;
            this.estadosEM = estadosEMi;
            this.estadosCB = estadosCBi;
            this.estadosEB = estadosEBi;

    }

    public void adicionaEstado(EstadoAuxiliar s) {
            estados.add(s);
    }

    public void adicionaEstadoInicial(EstadoAuxiliar s) {
            estadosIniciais.add(s);
    }

    public void adicionaTransicao(EstadoAuxiliar p, EstadoAuxiliar q) {

            Set<EstadoAuxiliar> estadoReachable = transicoes.get(p);

            if (estadoReachable == null) {
                    estadoReachable = new HashSet<EstadoAuxiliar>();
                    estadoReachable.add(q);
                    transicoes.put(p, estadoReachable);
            } else {
                    if (!estadoReachable.contains(q)) {
                            estadoReachable.add(q);
                    }
            }

    }

    public String paraFormatoNuSMV() {

            String nl = System.getProperty("line.separator");
```

```
55              String spec = "";

                spec += "MODULE main" + nl;
                spec += "  VAR" + nl;
                spec += "    EstadoMC:{";
60
                Iterator<String> it1 = estadosMC.iterator();
                while (it1.hasNext()) {
                        String s = it1.next();
                        spec += s;
65                      if (it1.hasNext()) {
                                spec += ", ";
                        }
                }
                spec += "};" + nl;
70              spec += "    ControlePaletas:{";
                Iterator<String> it2 = estadosEM.iterator();
                while (it2.hasNext()) {
                        String s = it2.next();
                        spec += s;
75                      if (it2.hasNext()) {
                                spec += ", ";
                        }
                }
                spec += "};" + nl;
80              spec += "    EstadoMotor:{";
                Iterator<String> it3 = estadosCP.iterator();
                while (it3.hasNext()) {
                        String s = it3.next();
                        spec += s;
85                      if (it3.hasNext()) {
                                spec += ", ";
                        }
                }
                spec += "};" + nl;
90              spec += "    ControleBicos:{";
                Iterator<String> it4 = estadosCB.iterator();
                while (it4.hasNext()) {
                        String s = it4.next();
                        spec += s;
95                      if (it4.hasNext()) {
                                spec += ", ";
                        }
                }
```

```
                spec += "};" + nl;
100             spec += "    EstadoBombaDagua:{";
                Iterator<String> it5 = estadosEB.iterator();
                while (it5.hasNext()) {
                        String s = it5.next();
                        spec += s;
105                     if (it5.hasNext()) {
                                spec += ", ";
                        }
                }
                spec += "};" + nl;
110             spec += "    time: 1 .. " + maxInstant / 5 + ";" + nl;
                spec += "  INIT" + nl;

                Iterator<EstadoAuxiliar> itIS = estadosIniciais.iterator();
                while (itIS.hasNext()) {
115                     EstadoAuxiliar s = itIS.next();

                        spec += "    (EstadoMC = " + s.getName();
                        spec += " & ControlePaletas = " + s.getName2();
                        spec += " & EstadoMotor = " + s.getName3();
120                     spec += " & ControleBicos = " + s.getName4();
                        spec += " & EstadoBombaDagua = " + s.getName5();

                        if (!ignoreTime) {
                                spec += " & time = " + s.getTime();
125                     }

                        spec += ")";

                        if (itIS.hasNext()) {
130                             spec += "|";
                        }

                        spec += nl;
                }
135
        ...

                return spec;
        }
140 }
```

A classe **Principal**, inicializará o processo de criação do modelo reduzido do programa, através da criação dos objetos entradaLog (do tipo **EntradaLog**) e aplicaAlgorit (do tipo

AlgoritmoMerge). O objeto aplicaAlgorit, recebe como entrada as informações do *log* tratadas e, através da aplicação do método merge(), a variável espacoEstadosRed, do tipo **EspacoEstados**, é criada contendo a descrição do modelo reduzido do programa.

O modelo reduzido é então enviado para o objeto saidaModelo, do tipo **SaidaModelo**, para a criação do arquivo *log.smv* com a formatação enviada pelo objeto espacoEstadosRed, através do método paraFormatoNuSMV().

Listagem 4.11: Classe Principal

```
public class Principal {

    public static void main(String[] args) {

        TraceMerger aplicaAlgorit = null;

        File logFile = new File("log.txt");
        File smvFile = new File("log.smv");

        try {
            // Cria objetos para arquivos de Entrada e Saída
            EntradaLog entradaLog = new EntradaLog(logFile);
            SaidaModelo saidaSMV = new SaidaModelo(smvFile);

            // Cria objeto aplicaAlgoritmo e envia as
               informações do log
            // tradadas.
            aplicaAlgorit = new TimePreservingMerger(entradaLog
               .getEstadosMC(),
                        entradaLog.getEstadosCP(),
                            entradaLog.getEstadosEM(),
                        entradaLog.getEstadosCB(),
                            entradaLog.getEstadosEB());

            for (List<String> trace : entradaLog.getTraces()) {
                aplicaAlgorit.addTrace(trace);
            }

            // Inicia merge
            EspacoEstados espacoEstadosRed = aplicaAlgorit.
               merge();

            // Formata espaço de Espaço de Estados no formato
               NuSMV e escreve no
            // arquivo log.smv.
            saidaSMV.write(espacoEstadosRed.paraFormatoNuSMV())
```

```
                    ;

                } catch (FileNotFoundException e) {
                    System.out.println("ERROR: A specified file has not
                        been found.");
                    e.printStackTrace();
                } catch (IOException e) {
                    System.out.println("ERROR: There was an IO error.")
                        ;
                    e.printStackTrace();
                }

        }

}
```

A classe **SaidaModelo**, implementa as funcionalidades para escrever no arquivo **.smv* (extensão do NuSMV), as informações geradas do modelo reduzido do programa, na formatação do NuSMV, conforme trecho de código apresentado na Listagem 4.12.

Listagem 4.12: Classe SaidaModelo

```
public class SaidaModelo {

  private File arquivoSMV;

  public SaidaModelo(File output){
    this.arquivoSMV = output;
  }

  public void write(String text) throws IOException{
    FileWriter writer = new FileWriter(arquivoSMV);
    BufferedWriter buf = new BufferedWriter(writer);
    buf.write(text);
    buf.flush();
  }
}
```

Na Figura 4.6, apresenta-se a representação do modelo reduzido do programa no formato do *NuSMV*, construído a partir do *log* gerado e apresentado na Figura 4.4.

```
MODULE main
 VAR
  EstadoMC :{EjetandoAgua, Ocioso, Esperando, Limpando};
  ControlePaletas :{2, 1, 0};
  EstadoMotor :{2, 1, 0};
  ControleBicos :{true, false};
  EstadoBombaDagua :{true, false};
  time: 1 .. 5;
 INIT
  (EstadoMC = Limpando & ControlePaletas = 1 & EstadoMotor = 1 & ControleBicos = false & EstadoBombaDagua = false & time = 1)|
  (EstadoMC = Limpando & ControlePaletas = 2 & EstadoMotor = 2 & ControleBicos = false & EstadoBombaDagua = false & time = 1)|
  (EstadoMC = Limpando & ControlePaletas = 1 & EstadoMotor = 1 & ControleBicos = false & EstadoBombaDagua = false & time = 1)|
  (EstadoMC = Limpando & ControlePaletas = 2 & EstadoMotor = 2 & ControleBicos = false & EstadoBombaDagua = false & time = 1)|
  (EstadoMC = Esperando & ControlePaletas = 1 & EstadoMotor = 1 & ControleBicos = false & EstadoBombaDagua = false & time = 1)
 TRANS
  (EstadoMC = Ocioso & next(EstadoMC) = Ocioso & ControlePaletas = 0 & next(ControlePaletas) = 0 & EstadoMotor = 0 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Ocioso & next(EstadoMC) = Ocioso & ControlePaletas = 0 & next(ControlePaletas) = 0 & EstadoMotor = 0 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Ocioso & next(EstadoMC) = Ocioso & ControlePaletas = 0 & next(ControlePaletas) = 0 & EstadoMotor = 0 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)|
  (EstadoMC = Ocioso & next(EstadoMC) = Limpando & ControlePaletas = 0 & next(ControlePaletas) = 2 & EstadoMotor = 0 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)|
  (EstadoMC = EjetandoAgua & next(EstadoMC) = Esperando & ControlePaletas = 1 & next(ControlePaletas) = 0 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = true & next(ControleBicos) = false & EstadoBombaDagua = true & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = EjetandoAgua & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas) = 2 & EstadoMotor = 2 & next(EstadoMotor) = 2 &
ControleBicos = true & next(ControleBicos) = false & EstadoBombaDagua = true & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = EjetandoAgua & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas) = 1 & EstadoMotor = 2 & next(EstadoMotor) = 1 &
ControleBicos = true & next(ControleBicos) = false & EstadoBombaDagua = true & next(EstadoBombaDagua) = false & time = 2 & next(time) = 3)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 1 & next(ControlePaletas) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 2 & next(time) = 3)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 1 & next(ControlePaletas) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 1 & next(time) = 2)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 0 & next(ControlePaletas) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)|
  (EstadoMC = Esperando & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 2 & next(ControlePaletas) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas) = 1 & EstadoMotor = 2 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 1 & next(time) = 2)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 1 & next(ControlePaletas) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 1 & next(time) = 2)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas) = 2 & EstadoMotor = 2 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 1 & next(time) = 2)|
  (EstadoMC = Limpando & next(EstadoMC) = EjetandoAgua & ControlePaletas = 1 & next(ControlePaletas) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos) = true & EstadoBombaDagua = false & next(EstadoBombaDagua) = true & time = 1 & next(time) = 2)|
  (EstadoMC = Limpando & next(EstadoMC) = Ocioso & ControlePaletas = 2 & next(ControlePaletas) = 0 & EstadoMotor = 2 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = Limpando & next(EstadoMC) = Ocioso & ControlePaletas = 1 & next(ControlePaletas) = 0 & EstadoMotor = 1 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = Limpando & next(EstadoMC) = EjetandoAgua & ControlePaletas = 2 & next(ControlePaletas) = 1 & EstadoMotor = 2 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos) = true & EstadoBombaDagua = false & next(EstadoBombaDagua) = true & time = 2 & next(time) = 3)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas) = 2 & EstadoMotor = 2 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 2 & next(time) = 3)|
  (EstadoMC = Limpando & next(EstadoMC) = EjetandoAgua & ControlePaletas = 1 & next(ControlePaletas) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos) = true & EstadoBombaDagua = false & next(EstadoBombaDagua) = true & time = 2 & next(time) = 3)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas) = 2 & EstadoMotor = 2 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 1 & next(ControlePaletas) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Limpando & next(EstadoMC) = Ocioso & ControlePaletas = 2 & next(ControlePaletas) = 0 & EstadoMotor = 2 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas) = 1 & EstadoMotor = 2 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)
```

Figura 4.6: Representação do modelo gerado

4.2.3 Verificação do modelo e correção do código

Na verificação do modelo gerado é necessário descrever os requisitos que deseja-se verificar em linguagem LTL ou CTL. Os possíveis estados do módulo controlador apresentados na Figura 4.2, são construídos a partir da especificação apresentada na Seção 4.1.1, na qual verifica-se o seguinte requisito:

O estado do módulo controlador ***Ejetando Água*** *sempre é seguido do estado* ***Esperando*** *ou* ***Ejetando Água***.

Utiliza-se a formatação do NuSMV para escrever o requisito em linguagem CTL, conforme Equação 4.1 (27).Esta expressão deve ser adicionada no final do arquivo do modelo gerado, para formar o arquivo de entrada do verificador de modelos NuSMV.

$$SPECAG(EstadoMC = EjetandoAgua \rightarrow AX(EstadoMC = Esperando|EstadoMC = EjetandoAgua)) \tag{4.1}$$

Com o modelo e a especificação sendo testados no mesmo arquivo, o verificador de modelo NuSMV é executado. Na Figura 4.7, apresenta-se o resultado da verificação. Como pode ser observado, um contra-exemplo é gerado para o requisito da especificação testada, visto que existe um traço que permite o estado **Limpando** após o estado **EjetandoAgua**.

```
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification AG (EstadoMC = EjetandoAgua -> AX (EstadoMC = Esperando | EstadoMC = EjetandoAgua)) is false

-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  EstadoMC = Limpando
  ControlePaletas = 1
  EstadoMotor = 1
  ControleBicos = false
  EstadoBombaDagua = false
  time = 1
-> Input: 1.2 <-
-> State: 1.2 <-
  EstadoMC = EjetandoAgua
  ControlePaletas = 2
  EstadoMotor = 2
  ControleBicos = true
  EstadoBombaDagua = true
  time = 2
-> Input: 1.3 <-
-> State: 1.3 <-
  EstadoMC = Limpando
  ControlePaletas = 1
  EstadoMotor = 1
  ControleBicos = false
  EstadoBombaDagua = false
  time = 3
```

Figura 4.7: Contra-exemplo encontrado pelo verificador de modelos NuSMV

Com o contra-exemplo gerado, é possível observar que existe um erro no código fonte, quando o estado **Limpando** é atribuído ao módulo controlador. Analisando a listagem

4.13, as condições de teste do **if**, que atribui o estado **Limpando** ao módulo controlador, é necessário também verificar que o estado atual não é **EjetandoAgua**.

Listagem 4.13: Classe Controlador Corrigida

```
if ((controlePaletas > 0) & (controleBicos == false)
                & (estadoModuloControlador != "EjetandoAgua")
                & (estadoModuloControlador != "Esperando")) {
        estadoModuloControlador = "Limpando";
        saidasMC.setEstadoBombaDagua(false);
        saidasMC.setEstadoMotores(controlePaletas);
        salvaEstado();
}
```

Após esta modificação no código fonte, um novo arquivo de *log* deve ser gerado para que um novo modelo reduzido do programa seja obtido e novamente testado. Na Figura 4.8, observa-se o *log* obtido após compilação e execução do programa com as modificações apresentadas na listagem 4.13. A partir do *log* gerado, o espaço de estado no formato do NuSMV é gerado novamente, conforme ilustrado na Figura 4.9.

```
EstadoMC
EjetandoAgua Ocioso Esperando Limpando
ControlePaletas
2 1 0
EstadoMotor
2 1 0
ControleBicos
false true
EstadoBombaDagua
false true

TRACES
Limpando 2 2 false false Ocioso 0 0false false Limpando 1 1false false Limpando 2 2false false Limpando 1 1false false
Limpando 2 2 false false EjetandoAgua 1 1 true true Esperando 0 1false false Esperando 2 1false false Esperando 1 1false false
Ocioso 0 0 false false Limpando 1 1false false Limpando 1 1false false Ocioso 0 0false false Limpando 1 1false false
Ocioso 0 0 false false EjetandoAgua 1 1 true true Esperando 1 1false false Esperando 0 1false false Esperando 0 1false false
Esperando 1 1 false false Esperando 2 1false false Esperando 1 1false false Ocioso 0 0false false Limpando 1 1false false
```

Figura 4.8: Log gerado após a correção do código fonte

```
MODULE main
 VAR
  EstadoMC :{EjetandoAgua, Ocioso, Esperando, Limpando};
  ControlePaletas :{2, 1, 0};
  EstadoMotor :{2, 1, 0};
  ControleBicos :{true, false};
  EstadoBombaDagua :{true, false};
  time: 1 .. 5;
 INIT
  (EstadoMC = Ocioso & ControlePaletas = 0 & EstadoMotor = 0 & ControleBicos = false & EstadoBombaDagua = false & time = 1)|
  (EstadoMC = Ocioso & ControlePaletas = 0 & EstadoMotor = 0 & ControleBicos = false & EstadoBombaDagua = false & time = 1)|
  (EstadoMC = Limpando & ControlePaletas = 2 & EstadoMotor = 2 & ControleBicos = false & EstadoBombaDagua = false & time = 1)|
  (EstadoMC = Limpando & ControlePaletas = 2 & EstadoMotor = 2 & ControleBicos = false & EstadoBombaDagua = false & time = 1)|
  (EstadoMC = Esperando & ControlePaletas = 1 & EstadoMotor = 1 & ControleBicos = false & EstadoBombaDagua = false & time = 1)
 TRANS
  (EstadoMC = Ocioso & next(EstadoMC) = Limpando & ControlePaletas = 0 & next(ControlePaletas ) = 1 & EstadoMotor = 0 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)|
  (EstadoMC = Ocioso & next(EstadoMC) = Limpando & ControlePaletas = 0 & next(ControlePaletas ) = 1 & EstadoMotor = 0 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)|
  (EstadoMC = Ocioso & next(EstadoMC) = Limpando & ControlePaletas = 0 & next(ControlePaletas ) = 1 & EstadoMotor = 0 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 2 & next(time) = 3)|
  (EstadoMC = Ocioso & next(EstadoMC) = EjetandoAgua & ControlePaletas = 0 & next(ControlePaletas ) = 1 & EstadoMotor = 0 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = true & EstadoBombaDagua = false & next(EstadoBombaDagua) = true & time = 1 & next(time) = 2)|
  (EstadoMC = Ocioso & next(EstadoMC) = Limpando & ControlePaletas = 0 & next(ControlePaletas ) = 1 & EstadoMotor = 0 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 1 & next(time) = 2)|
  (EstadoMC = EjetandoAgua & next(EstadoMC) = Esperando & ControlePaletas = 1 & next(ControlePaletas ) = 0 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = true & next(ControleBicos ) = false & EstadoBombaDagua = true & next(EstadoBombaDagua) = false & time = 2 & next(time) = 3)|
  (EstadoMC = EjetandoAgua & next(EstadoMC) = Esperando & ControlePaletas = 1 & next(ControlePaletas ) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = true & next(ControleBicos ) = false & EstadoBombaDagua = true & next(EstadoBombaDagua) = false & time = 2 & next(time) = 3)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 2 & next(ControlePaletas ) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 2 & next(time) = 3)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 1 & next(ControlePaletas ) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 1 & next(time) = 2)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 2 & next(ControlePaletas ) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 0 & next(ControlePaletas ) = 0 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 0 & next(ControlePaletas ) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 1 & next(ControlePaletas ) = 0 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = Esperando & next(EstadoMC) = Ocioso & ControlePaletas = 1 & next(ControlePaletas ) = 0 & EstadoMotor = 1 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 0 & next(ControlePaletas ) = 0 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Esperando & next(EstadoMC) = Esperando & ControlePaletas = 1 & next(ControlePaletas ) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Limpando & next(EstadoMC) = EjetandoAgua & ControlePaletas = 2 & next(ControlePaletas ) = 1 & EstadoMotor = 2 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = true & EstadoBombaDagua = false & next(EstadoBombaDagua) = true & time = 1 & next(time) = 2)|
  (EstadoMC = Limpando & next(EstadoMC) = Ocioso & ControlePaletas = 2 & next(ControlePaletas ) = 0 & EstadoMotor = 2 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 1 & next(time) = 2)|
  (EstadoMC = Limpando & next(EstadoMC) = Ocioso & ControlePaletas = 1 & next(ControlePaletas ) = 0 & EstadoMotor = 1 & next(EstadoMotor) = 0 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 1 & next(ControlePaletas ) = 2 & EstadoMotor = 1 & next(EstadoMotor) = 2 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 3 & next(time) = 4)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 1 & next(ControlePaletas ) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 2 & next(time) = 3)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 1 & next(ControlePaletas ) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 1 & next(ControlePaletas ) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 1 & next(ControlePaletas ) = 1 & EstadoMotor = 1 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 5 & next(time) = 5)|
  (EstadoMC = Limpando & next(EstadoMC) = Limpando & ControlePaletas = 2 & next(ControlePaletas ) = 1 & EstadoMotor = 2 & next(EstadoMotor) = 1 &
ControleBicos = false & next(ControleBicos ) = false & EstadoBombaDagua = false & next(EstadoBombaDagua) = false & time = 4 & next(time) = 5)

  SPEC AG (( EstadoMC = EjetandoAgua) -> AX(( EstadoMC = Esperando)))
```

Figura 4.9: Descrição do modelo gerado após correção do código fonte

Com o novo modelo gerado o verificador de modelos *NuSMV* é executado novamente para a verificação do requisito da especificação. Conforme resultado apresentado na Figura 4.10, o modelo gerado não apresentou a falha. De forma que o verificador de modelos não apresentou um contra-exemplo para a especificação em teste.

```
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification AG (EstadoMC = EjetandoAgua -> AX (EstadoMC = Esperando | EstadoMC = EjetandoAgua)) is true
```

Figura 4.10: Verificação do modelo gerado após correção do código fonte

Capítulo 5

Considerações Finais e Trabalhos Futuros

Nesta dissertação é introduzido um método para aumentar a confiança no funcionamento de programas que não haviam sido submetidos a nenhum tipo de verificação formal. Para este fim, um modelo reduzido do programa foi construído a partir das informações coletadas de múltiplos traços de execuções do programa e da aplicação de um algoritmo. O modelo e as especificações do programa foram escritos na linguagem do verificador de modelos NuSMV, que é a ferramenta utilizada para a verificação do modelo. Esta ferramenta apresentou os contra-exemplos das especificações que falharam a partir de traços do programa. Com a análise destes traços, os erros no código foram rastreados, detectados e corrigidos.

O método se destaca por apresentar uma automatização da implementação, visto que os traços das execuções, a construção do modelo reduzido do programa e a formatação para a linguagem da ferramenta NuSMV ocorrem de forma automática. Desta forma, a complexidade de construção do modelo para a ferramenta NuSMV é ocultada, sendo necessário apenas a descrição das especificações do programa em LTL ou CTL de acordo com a semântica do NuSMV.

As principais dificuldades encontradas neste trabalho foram relacionadas a adaptação de algoritmos existentes para a construção de modelos formados por múltiplas variáveis e para traços de comprimentos diferentes. Outra dificuldade encontrada está relacionada a quantidade de traços que deve ser coletada para a construção do modelo. Neste trabalho, utiliza-se as restrições de tempo de processamento e espaço para armazenamento do arquivo de *log* como limitantes para a determinação da quantidade de traces a ser coletada, visto que a solução para a determinação de um valor ótimo, não fez parte do escopo desse trabalho.

5.1 Trabalhos Futuros

A partir do presente trabalho, pode-se delinear as seguintes propostas de pesquisa:

- Adaptação do algoritmo para verificar problemas de programas em arquiteturas multi-core;
- Desenvolver um tradutor que escreva automaticamente as especificação do sistema para as linguagens LTL ou CTL;
- Aplicar a técnica apresentada para outros estudos de caso que envolvem problemas de outras áreas, tendo como objetivo obter uma maior confiança com relação ao procedimento proposto;
- Investigar formas alternativas para o armazenamento dos traços do programa, por exemplo, através da utilização de banco de dados;
- Tratar o problema da incerteza relacionada a quantidade recomendada de traços do programa que devem ser coletados para que tenhamos uma boa aproximação do modelo ideal;

Referências Bibliográficas

1 BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: *FOSE '07: 2007 Future of Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2007. p. 85–103. ISBN 0-7695-2829-5.

2 GROSS, H.-G. *Component-Based Software Testing with UML.* Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 3642058825, 9783642058820.

3 ENGEL, A.; LAST, M. Modeling software testing costs and risks using fuzzy logic paradigm. *Journal of Systems and Software*, v. 80, n. 6, p. 817 – 835, 2007. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/B6V0N-4M69JBT-1/2/6c8954df3e3c93ebcd51b88d6e7483bb>.

4 UTTING, M.; LEGEARD, B. *Practical Model-Based Testing: A Tools Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN 0123725011.

5 PRETSCHNER, A. et al. One evaluation of model-based testing and its automation. In: *ICSE '05: Proceedings of the 27th international conference on Software engineering.* New York, NY, USA: ACM, 2005. p. 392–401. ISBN 1-59593-963-2.

6 LUIZ, S. O. D.; VASCONCELOS, G. de M.; SILVA, L. D. da. Formal specification of dsp gateway for data transmission between processor cores of omap platform. In: *Proceedings of the 2008 ACM symposium on Applied computing.* New York, NY, USA: ACM, 2008. (SAC '08), p. 1545–1549. ISBN 978-1-59593-753-7. Disponível em: <http://doi.acm.org/10.1145/1363686.1364046>.

7 KANSTRéN ERIC PIEL, H. G. G. T. Trace-based code generation for model-based testing. In: . Denver, Colorado, USA: Eighth International Conference on Generative Programming and Component Engineering, 2009.

8 VASCONCELOS, G.; PERKUSICH, A.; ALMEIDA, H. An architecture for the simulation of pervasive applications for automobiles. In: *XVI Congresso e Exposição Internacionais de Tecnologia da Mobilidade.* [S.l.: s.n.], 2007.

9 GEILEN, M. C. On the construction of monitors for temporal logic properties. In: HAVELUND, K.; ROŞU, G. (Ed.). *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)*. [S.l.]: Elsevier Science, 2001. (Electronic Notes in Theoretical Computer Science, 2).

10 MEYER, B. Applying "design by contract". *IEEE COMPUTER*, v. 25, p. 40–51, 1992.

11 BARTETZKO, D. et al. *Jass - Java with Assertions*. 2001.

12 AMMONS, G.; BODIK, R.; LARUS, J. R. Mining specifications. In: . [S.l.: s.n.], 2002. p. 4–16.

13 LO, D.; KHOO, S. cheng; LIU, C. Mining temporal rules from program execution traces. *Int. Work. on Prog. Comprehension*, 2007.

14 ERNST, M. D. et al. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 69, n. 1-3, p. 35–45, 2007. ISSN 0167-6423.

15 BOSHERNITSAN, M.; DOONG, R.; SAVOIA, A. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In: . [S.l.]: ACM Press, 2006. p. 169–180.

16 PACHECO, C.; ERNST, M. D. Eclat: Automatic generation and classification of test inputs. In: *In 19th European Conference Object-Oriented Programming*. [S.l.: s.n.], 2005. p. 504–527.

17 XIE, T.; NOTKIN, D. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, v. 2006, 2006.

18 LORENZOLI, D.; MARIANI, L.; PEZZè, M. Automatic generation of software behavioral models. In: *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. p. 501–510. ISBN 978-1-60558-079-1.

19 CHO, H. et al. Algorithms for approximate fsm traversal. In: *DAC '93: Proceedings of the 30th international Design Automation Conference*. New York, NY, USA: ACM, 1993. p. 25–30. ISBN 0-89791-577-1.

20 SILVA, P. S. da; MELO, A. C. V. de. Model checking merged program traces. *Electron. Notes Theor. Comput. Sci.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 240, p. 97–112, 2009. ISSN 1571-0661.

21 JR., E. M. C.; GRUMBERG, O.; PELED, D. A. *Model Checking.* [S.l.]: The MIT Press, 1999.

22 LICHTENSTEIN, O.; PNUELI, A. Checking that finite state concurrent programs satisfy their linear specification. In: *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* New York, NY, USA: ACM, 1985. p. 97–107. ISBN 0-89791-147-4.

23 VARDI, M. Y.; WOLPER, P. An automata-theoretic approach to automatic program verification. In: *LICS.* [S.l.: s.n.], 1986. p. 332–344.

24 BERTOLI, P.; CIMATTI, A.; ROVERI, M. Heuristic search + symbolic model checking = efficient conformant planning. In: . [S.l.]: AAAI Press, 2001. p. 467–472.

25 CIMATTI, A. et al. Nusmv: A new symbolic model verifier. In: HALBWACHS, N.; PELED, D. (Ed.). [S.l.]: Springer, 1999. (Lecture Notes in Computer Science), p. 495–499.

26 MCMILLAN, K. L. *Symbolic model checking: an approach to the state explosion problem.* Tese (Doutorado), Pittsburgh, PA, USA, 1992.

27 CAVADA ALESSANDRO CIMATTI, C. A. J. R.; TCHALTSEV, A. *NuSMV 2.4 User Manual.* [S.l.], 2005.

28 FRASER, G.; WOTAWA, F.; AMMANN, P. Issues in using model checkers for test case generation. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 82, n. 9, p. 1403–1418, 2009. ISSN 0164-1212.

29 FRASER, G.; WOTAWA, F. Using ltl rewriting to improve the performance of model-checker based test-case generation. In: *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing.* New York, NY, USA: ACM, 2007. p. 64–74. ISBN 978-1-59593-850-3.