PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GERAÇÃO DE CONTRAEXEMPLOS E TESTEMUNHAS PARA UM VERIFICADOR DE MODELOS DESCRITOS EM REDES DE AUTÔMATOS ESTOCÁSTICOS

CLAITON MARQUES CORREA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Fernando Luís Dotti

Porto Alegre
2013

Dados Internacionais de Catalogação na Publicação (CIP)

C824g Correa, Claiton Marques
Geração de contraexemplos e testemunhas para um verificador de modelos descritos em redes de autômatos estocásticos / Claiton Marques Correa. – Porto Alegre, 2013.
107 p.

Diss. (Mestrado em Ciência da Computação) – Faculdade de Informática, PUCRS.
Orientação: Prof. Dr. Fernando Luís Dotti.

1. Informática. 2. Redes de Autômatos Estocásticos. 3. Simulação E Modelagem Em Computadores. I. Dotti, Fernando Luís. II. Título.

CDD 003.3

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "Geração de Contra – Exemplos e Testemunhas para um Verificador de Modelos Descritos em Redes de Autômatos Estocásticos" apresentada por Claiton Marques Correa como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 26/03/2013 pela Comissão Examinadora:

Prof. Dr. Fernando Luís Dotti – Orientador — PPGCC/PUCRS

Prof. Dr. Paulo Henrique Lemelle Fernandes – PPGCC/PUCRS

Prof. Dr. Afonso Henrique Corrêa de Sales – FACIN/PUCRS

Prof. Dr. Lucio Mauro Duarte – UFRGS

Homologada em 25/06/2013, conforme Ata No. 011 pela Comissão Coordenadora.

Prof. Dr. Paulo Henrique Lemelle Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

DEDICATÓRIA

Dedico este trabalho a meus pais.

"Não existem pessoas de sucesso e pessoas fracassadas. O que existe são pessoas que lutam por seus sonhos ou desistem deles."
(Augusto Cury)

AGRADECIMENTOS

Chega ao fim mais uma etapa em minha vida. Conclui o mestrado e como diria Galvão Bueno: é teeetraaaaa!!! É teetraaaa!!!

Durante alguns dias, enquanto escrevia a dissertação, fiquei pensando no que escrever nesta seção de agradecimentos. Creio que esta seja uma parte do trabalho que as pessoas sempre leem, visto que é nela que os autores podem escrever e expressar com maior liberdade, sem preocupações com formalismos e correções, um pouco do que vivenciaram durante a condução do trabalho.

É difícil não olhar para o passado e tentar traçar o caminho percorrido até aqui, como também é difícil não tentar imaginar o que virá pela frente, sonhar. Contudo, sonhar não é suficiente, há de se agir para por em prática planos e desejos.

Nestes dois últimos anos vivi situações que me proporcionaram ensinamentos para vida. Sem sombra de dúvidas fazer um mestrado não é uma tarefa fácil, é um desafio. Eu seria pretencioso se pensasse que cheguei até aqui sozinho, ninguém faz um mestrado sozinho, digo mais, nenhum ser humano faz algo sozinho. Desse modo, poderia fazer uma lista enorme com os nomes das pessoas que me ajudaram, desde ingressar no mestrado, até concluí-lo.

Ainda que difícil, é necessário mencionar algumas dessas pessoas. Gostaria de agradecer ao meu orientador, Dr. Fernando Luís Dotti, pela compreensão, pela aposta em mim. Agradeço pelo seu empenho, pois nestes dois anos, mesmo com todas as responsabilidades e compromissos como coordenador da pós-graduação e logo depois diretor da Faculdade de Informática, dedicaste sempre algumas horas das tuas semanas para a orientação de teus alunos. Tens meu respeito e admiração.

Agradeço também aos professores Paulo Fernandes e Afonso Sales pela colaboração com observações e sugestões para este trabalho.

Conheci pessoas incríveis em Porto Alegre. Mas, antes de falar delas, quero destacar a parceria e amizade (desde a época de graduação) que ao longo do mestrado esteve presente. Anderson Borges, Fabio Quintana, Guilherme Benites, Joaquim Assunção e Raphael Bohrer, colegas e amigos desde a graduação na saudosa PUCRS Uruguaiana, que sempre estiveram por perto, meu agradecimento pela amizade, por compreender os muitos finais de semana em que tive de abrir mão da companhia de vocês para tocar este trabalho.

Como disse acima, conheci pessoas incríveis em Porto Alegre. Algumas delas aqui da capital, outras, assim como eu, do interior do estado. Como não lembrar da Luciana Espindola corrigindo os primeiros textos que eu escrevia, pondo as mãos à cabeça e entre boas gargalhadas dizendo: "Claiton, isso que tu escreveu não faz sentido!!!". Hueheueh, tri divertido. Além dos ensinamentos

para vida, levo desse mestrado um sotaque aos pedaços, resultado da influência de colegas de diversos lugares do estado e país, mas principalmente pela influência do Bernardo Estácio, não fosse a tentativa desse paraense em imitar o jeito gaúcho de falar, hoje eu não estaria pronunciando /chocolat/, /merthiolat/, /abacat/ e por ai vai...

Brincadeiras à parte, quero agradecer a todos, meus colegas de Paloeprospec e os demais colegas de Universidade do PLN, GRV, GMAP, LABIO, GPIN, LAD e PET-Info. A todos vocês o meu muito obrigado.

Agradeço também ao Valmor, grande amigo dos meus pais, e sua família pela acolhida na capital. Valeu pela força.

Por último, nem por isso menos importante, agradeço aos meus pais pelo apoio emocional e financeiro. Obrigado por compreenderem minha constante ausência nestes dois anos de mestrado.

GERAÇÃO DE CONTRAEXEMPLOS E TESTEMUNHAS PARA UM VERIFICADOR DE MODELOS DESCRITOS EM REDES DE AUTÔMATOS ESTOCÁSTICOS *

RESUMO

A possibilidade de geração de contraexemplos e testemunhas é um dos principais atrativos da técnica de Verificação de Modelos. Os contraexemplos são uma boa fonte para depuração do sistema, pois são gerados quando uma especificação é refutada pelo modelo. Já as testemunhas ratificam a satisfação de uma especificação pelo modelo através de uma execução do sistema. Esta dissertação de Mestrado é parte de um projeto de construção de um verificador de modelos para modelos descritos em Redes de Autômatos Estocásticos e trata da implementação da geração de contraexemplos e testemunhas para a ferramenta.

Palavras-Chave: contraexemplos, testesmunhas, Redes de Autômatos Estocásticos, Verificação de Modelos.

*Trabalho parcialmente suportado pelos projetos FAPERGS PqG 1014867 e CNPq 560036/2010-8 "Verificação de Modelos Descritos em Redes de Autômatos Estocásticos".

COUNTEREXAMPLES AND WITNESSES GENERATION FOR THE STOCHASTIC AUTOMATA NETWORKS MODEL CHECKER *

ABSTRACT

The counterexamples and witnesses generation is one of the main attractive features of Model Checking. Counterexamples are a great data source to debug the system, because they are generated when a specification is violeted by a model of the system. On other hand, witnesses show that a model of the system holds for an specification, through an execution trace of the system. This dissertation is part of a project aimed to the construction of a Model Checker for Stochastic Automata Networks and focuses in the generation of counterexamples and witnesses for the tool.

Keywords: counterexamples, witnesses, Stochastic Automata Networks, Model Checking.

* This work is partially sponsored by research projects FAPERGS PqG 1014867 and CNPq 560036/2010-8 "Model Checking Stochastic Automata Networks".

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ALMC – The Local Model Checking Algorithm

CTL – Computation Tree Logic

EMC – Extended Model Checking

ENF – Existential Normal Form

LAD – Laboratório de Alto Desempenho

LTL – Linear Temporal Logic

MDD – Multivalued Decision Diagrams

PSS – Product State Space

RSS – Reachable State Space

SAN – Stochastic Automata Networks

SCC – Strongly Connected Component

LISTA DE ALGORITMOS

SUMÁRIO

1. INTRODUÇÃO

Um sistema está correto quando ele implementa o serviço definido como sua função [1]. Sistemas baseados em *software* são utilizados para controlar os mais diversos tipos de serviços, desde o computador de bordo de um automóvel até o controlador de voo de uma aeronave, onde por exemplo, falhas não são admitidas, uma vez que podem provocar a perda de inúmeras vidas humanas. A demanda por sistemas cada vez mais complexos e robustos torna importante a verificação da correção destes sistemas, uma vez que, como mencionado, a falha de um *software* pode levar a perdas não só de ordem financeira, mas também humana. Porém, garantir que um sistema esteja correto não é uma tarefa fácil. Conforme Clarke *et. al* [12], mais tempo é gasto na verificação do que na construção de projetos de *hardware* e sistemas complexos.

Tendo em vista a necessidade de verificar a correção de sistemas e garantir sua confiabilidade, no decorrer dos últimos anos, importantes técnicas de verificação foram desenvolvidas. Entre elas, a técnica de verificação de modelos mostra-se atrativa por permitir a geração de contraexemplos e testemunhas como saídas para o usuário [2, 12, 14].

A técnica possui como entrada o sistema que será verificado, descrito por algum formalismo na forma de um sistema de transição de estados e a especificação que será avaliada, descrita em lógica temporal [26]. Quando o resultado da verificação é falso, ou seja, o modelo não atende à especificação avaliada, o verificador de modelos gera para o usuário um contraexemplo, que nada mais é do que uma sequência de estados e transições que mostra um *trace* de execução onde o sistema refuta aquela especificação. Por outro lado, quando o resultado da verificação é verdadeiro, ou seja, o sistema atende à especificação avaliada, o verificador pode gerar uma testemunha, isto é, uma sequência de estados e transições que mostra um *trace* de execução onde o sistema satisfaz a especificação avaliada. Segundo Clarke e Veith [14], muitas pessoas usam a verificação de modelos em razão da característica de geração de contraexemplos, pois eles têm um grande valor para depuração do sistema.

Em relação aos formalismos utilizados para modelagem de sistemas, Redes de Autômatos Estocásticos, SAN (*Stochastic Automata Networks*, em inglês) mostra-se atraente, pois, é um formalismo adequado para modelagem de sistemas e focado principalmente em avaliação de desempenho [28]. SAN tem sido utilizada para modelar diferentes tipos de sistemas, como: programas paralelos mestre/escravo [4], rede wireless adhoc [17], linha de produção [19], equipes de desenvolvimento de software [20], entre outros. Um diferencial de SAN perante outros formalismos markovianos usados para descrição de modelos de sistemas, reside no fato de o formalismo suportar o conceito de funções, que avaliam o estado global de uma rede de autômatos. O uso de funções pode descrever dependências entre autômatos e permite a descrição de comportamentos complexos de uma forma compacta [5].

SAN, ao contrário de outros formalismos, não possui uma ferramenta que dê suporte à verificação de seus modelos. Como exemplos de formalismos que possuem um verificador é possível citar: Cadeias de Markov com o PRISM [22], Redes de Petri com o SMART [8] e Álgebra

de Processos com o FDR (*Failures Divergence Refinement*) [6]. Uma ferramenta de suporte à verificação de modelos contribui, através da geração de contraexemplos, para detecção e então correção de erros pelos modeladores.

Este conjunto de fatores, ou seja, a importância de verificar a correção de sistemas, os atrativos da técnica de verificação de modelos, aliados ao fato de SAN não possuir uma ferramenta que verifique os diversos modelos já descritos com o formalismo, tornam relevantes os esforços em construir um verificador de modelos para SAN.

O trabalho apresentado ao longo desta dissertação está inserido dentro do projeto de construção de um verificador de modelos para o formalismo de Redes de Autômatos Estocásticos, verificador que recebeu o nome de SAN *Model Checker*, e tem como objetivo implementar a geração de contraexemplos e testemunhas para a ferramenta.

Ao longo do texto são apresentados ao leitor, além dos conceitos de compreensão necessária para o entendimento do trabalho, os primeiros resultados obtidos no sentido de construir um verificador para SAN [15], bem como a estratégia e algoritmos desenvolvidos que possibilitaram a implementação do recurso de contraexemplos e testemunhas para a ferramenta.

A bateria de testes utilizada para analisar e demonstrar o comportamento do algoritmo de geração de contraexemplos e testemunhas compreendeu a definição de um conjunto de 48 especificações, descritas em lógica temporal ramificada (*Computation Tree Logic*, em inglês). Algumas das especificações criadas possuem sentido semântico para os problemas apresentados, outras contudo, foram criadas apenas com o propósito de demonstrar o funcionamento do algoritmo implementado. Com a intenção de verificar a coerência dos resultados alcançados, as saídas geradas pelo SAN *Model Checker* para 18 das especificações criadas foram comparadas às saídas geradas pelo NuSMV *Model Checker* [9] para os mesmos modelos e especificações . Neste ponto, torna-se importante ressaltar que o SAN *Model Checker*, diferentemente do NuSMV, não faz uso de técnicas para geração de contraexemplos e testemunhas que gerem o resultado mais curto possível, no que se refere ao tamanho do *trace* de execução apresentado.

Por fim, o foco e motivação do trabalho que será apresentado no decorrer desta dissertação é a geração de contraexemplos e testemunhas para o verificador de modelos descritos em Redes de Autômatos Estocásticos. O trabalho é organizado da seguinte forma:

- O Capítulo 2 apresenta conceitos sobre Redes de Autômatos Estocásticos, verificação de modelos, lógicas temporais e contraexemplos e testemunhas, necessários para a condução deste trabalho.

- O Capítulo 3 apresenta a arquitetura do verificador de modelos descritos em Redes de Autômatos Estocásticos.

- O Capítulo 4 detalha a estratégia utilizada e os algoritmos desenvolvidos para implementação do recurso de contraexemplos e testemunhas na ferramenta.

- O Capítulo 5 mostra os experimentos realizados para demonstrar o funcionamento da ferramenta e a comparação dos resultados gerados por ela com os resultados gerados pelo NuSMV *Model Checker* [9].
- O Capítulo 6 faz as considerações finais do trabalho realizado e trata de trabalhos futuros viabilizados com o trabalho apresentado.

O trabalho também é composto por três apêndices, que são os modelos SAN em formato de texto utilizados ao longo da dissertação e nos experimentos realizados.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo é dedicado ao estudo de conceitos necessários para o entendimento e desenvolvimento do trabalho. Na Seção 2.1, o formalismo de Redes de Autômatos Estocásticos é apresentado, seguido por três exemplos de modelos utilizados ao longo da dissertação. A Seção 2.2 trata da técnica de verificação de modelos, utilizada para verificar a correção de sistemas. A técnica possui como entrada um sistema descrito por um formalismo, por exemplo, Redes de Autômatos Estocásticos, na forma de um sistema de transição de estados e uma especificação, descrita em lógica temporal. A Seção 2.3 introduz conceitos sobre lógica temporal, para então apresentar a lógica temporal ramificada (*Computation Tree Logic - CTL*, em inglês) e sua forma existencial normal (*Existential Normal Form - ENF*, em inglês). A Seção 2.4 apresenta os conceitos de contraexemplos e testemunhas para fórmulas CTL. Além disso, a seção também aborda conceitos de contraexemplos e testemunhas em árvore e contraexemplos e testemunhas para fórmulas CTL em ENF. Na Seção 2.5 são apresentados dois trabalhos relacionados a este: *Implementing a CTL Model Checker* [21] de Keijo Heljanko e *Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking* [11] de Clarke *et. al*.

2.1 Redes de Autômatos Estocásticos

Redes de Autômatos Estocásticos ou em inglês *Stochastic Automata Networks*, SAN, é um formalismo para modelagem de sistemas com grande espaço de estados [28]. Com SAN é possível modelar um sistema em vários subsistemas que interagem ocasionalmente.

Um modelo SAN é descrito por um conjunto de autômatos, cada qual com um conjunto de estados e transições. As transições em um autômato são associadas a eventos, locais ou sincronizantes, que disparam a transição. Os eventos locais alteram o estado de um autômato e permitem aos autômatos ter comportamentos paralelos. Já os eventos sincronizantes alteram o estado de dois ou mais autômatos simultaneamente. Eles proporcionam interação entre os autômatos pelo fato de existir sincronismo no disparo das transições.

2.1.1 Autômatos

Um autômato é um modelo matemático de um sistema, com entradas e saídas discretas. O sistema pode se encontrar em qualquer um dos estados que compõem esse sistema. A Figura 2.1 modela um sistema através de três autômatos. Cada autômato possui três estados: *Thinking*, *Left* e *Right*. O estado individual de cada autômato representa o seu estado local. A transição entre os estados de cada autômato pode ser local ou sincronizante. O estado global ou espaço de estado

produto de uma Rede de Autômatos Estocásticos é dado pela combinação dos estados locais de cada autômato pertencente à rede.

2.1.2 Transições Locais

As transições locais alteram o estado de um autômato, ou seja, alteram o espaço de estado produto do sistema através da mudança do estado de um autômato. Essas transições permitem aos autômatos ter um comportamento paralelo. Na Figura 2.1 todos os eventos são locais. É importante salientar que apesar dos eventos locais serem independentes, eles são disparados um de cada vez, visto que a escala de tempo contínuo, dois ou mais eventos não ocorrem ao mesmo tempo.

2.1.3 Transições Sincronizantes

As transições sincronizantes alteram o estado de dois ou mais autômatos simultaneamente. Da mesma forma que transições locais, elas alteram o espaço de estado produto do sistema. As transições sincronizadas proporcionam interação entre os autômatos pelo fato de existir sincronismo no disparo das transições. No exemplo da Figura 2.2 os eventos tx_5, $g_{1,2}$, $g_{2,3}$, $g_{3,4}$, $g_{4,5}$ e $g_{5,6}$ são sincronizantes.

2.1.4 Taxas e Probabilidades Funcionais

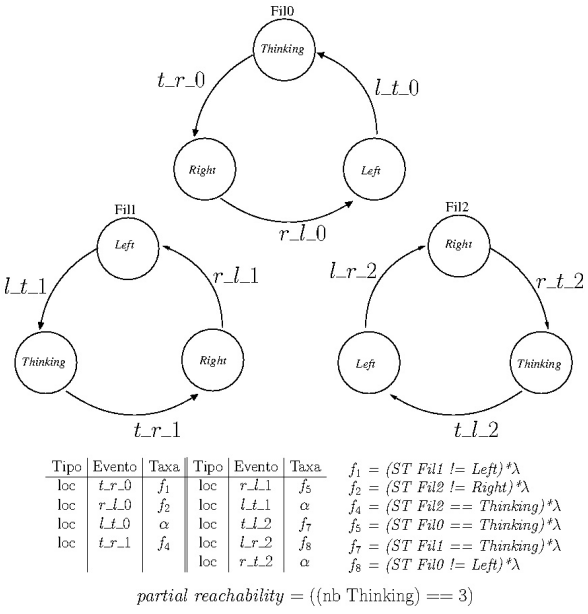
As taxas e probabilidades funcionais representam outra forma de representar interação entre os autômatos em uma rede e representam também a característica que difere SAN de outros formalismos como Redes de Petri [3] e Cadeias de Markov [32]. As taxas funcionais permitem o disparo ou não de um evento dependendo do estado de outro(s) autômato(s) referenciados(s) na função. Na Figura 2.1, o evento t_r_1 possui uma taxa funcional f_4 e só ocorrerá se o estado do filósofo dois (Fil2) é *Thinking*.

Para cada modelo deve ser definida uma função booleana de atingibilidade ou *partial reachability*, esta função determina respectivamente o conjunto ou subconjunto de estados atingíveis dentro do espaço total de estados do modelo. Quando esta função for igual a um, significa que todos os estados do espaço produto do modelo são atingíveis. Dessa forma temos o espaço de estado produto (*Product Space State - PSS*) que é o conjunto de todos os estados que fazem parte do modelo e o espaço de estado atingível (*Reachable Space State - RSS*) que compreende o conjunto de estados atingível no modelo.

Para o melhor entendimento de SAN, abaixo são brevemente descritos alguns modelos utilizados ao longo deste trabalho.

2.1.5 Jantar dos Filósofos

A Figura 2.1 ilustra o modelo SAN para o clássico problema do Jantar dos Filósofos para três filósofos. Os filósofos estão dispostos em uma mesa que possui o número de garfos e pratos igual ao número de filósofos sentados à ela. Eles podem estar em um dos três estados possíveis, que são: *Thinking*, *Right* ou *Left*. Para comer, um filósofo precisa de dois garfos, dessa forma, quando um filósofo está comendo, seus dois vizinhos não podem comer. No modelo representado pela Figura 2.1 quando um filósofo destro está *com fome* ele pega o garfo da direita, se for canhoto ele pega o da esquerda; representado na figura pelas transições *Thinking* para *Right*, para filósofos destros e *Thinking* para *Left* para o filósofo canhoto.



Tipo	Evento	Taxa	Tipo	Evento	Taxa
loc	t_r_0	f_1	loc	r_l_1	f_5
loc	r_l_0	f_2	loc	l_t_1	α
loc	l_t_0	α	loc	t_l_2	f_7
loc	t_r_1	f_4	loc	l_r_2	f_8
			loc	r_t_2	α

f_1 = *(ST Fil1 != Left)* $^{\sharp}\lambda$
f_2 = *(ST Fil2 != Right)* $^{\sharp}\lambda$
f_4 = *(ST Fil2 == Thinking)* $^{\sharp}\lambda$
f_5 = *(ST Fil0 == Thinking)* $^{\sharp}\lambda$
f_7 = *(ST Fil1 == Thinking)* $^{\sharp}\lambda$
f_8 = *(ST Fil0 != Left)* $^{\sharp}\lambda$

$partial\ reachability = ((\text{nb Thinking}) == 3)$

Figura 2.1 – Modelo SAN para o problema do Jantar dos Filósofos para três filósofos.

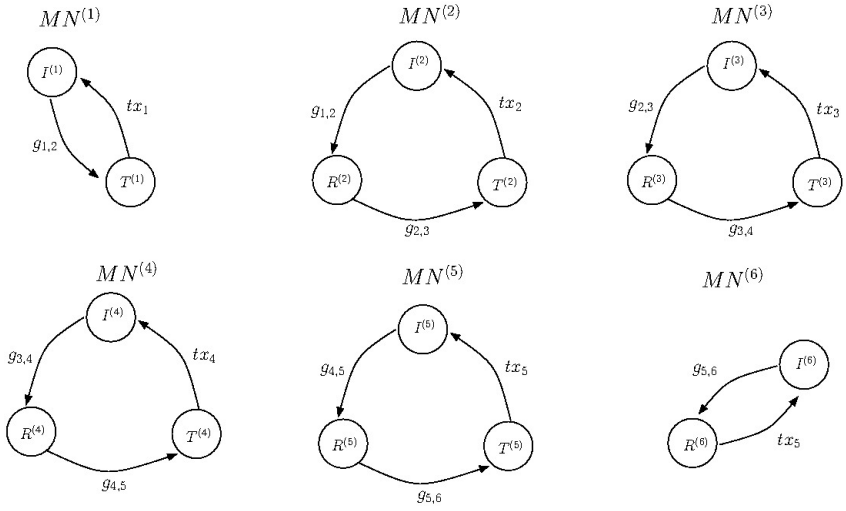
Para passar para o estado *comendo* os filósofos precisam pegar o segundo garfo. Essa mudança de estado para os filósofos destros é representada pela transição *Right* para *Left* e para o filósofo canhoto, pela transição *Left* para *Right*. Após comer, os filósofos voltam ao estado *Thinking*, representado na figura pelas transições *Left* para *Thinking*, para filósofos destros e *Right* para *Thinking* para o filósofo canhoto. No modelo representado pela Figura 2.1, apenas o último filósofo (Fil2) é canhoto.

As funções *st* e *nb* são utilizadas respectivamente para consultar o estado de um autômato e o número de autômatos em um determinado estado.

O exemplo da Figura 2.1 utiliza apenas eventos locais. As transições rotuladas pelos eventos l_t_0 , l_t_1 e r_t_2 são locais e não dependem do estado de algum outro autômato na rede. Por outro lado, todas as demais transições dependem da disponibilidade de um garfo e portanto são rotuladas com eventos que possuem dependências funcionais, por exemplo o evento

t_r_0 que possui a taxa funcional f_1 e somente ocorrerá se o estado do filósofo 1 (Fil1) for diferente de $Left$.

2.1.6 Modelo adhoc



$f_{1,2} = [((stMN^{(3)} == I^{(3)})\&\&(stMN^{(4)} == I^{(4)}))] * \lambda$

$f_{2,3} = [((stMN^{(1)} == I^{(1)})\&\&(stMN^{(4)} == I^{(4)})\&\&(stMN^{(5)} == I^{(5)})] * \lambda$

$f_{3,4} = [((stMN^{(1)} == I^{(1)})\&\&(stMN^{(2)} == I^{(2)})\&\&(stMN^{(5)} == I^{(5)})\&\&(stMN^{(6)} == I^{(6)}))] * \lambda$

$f_{4,5} = [((stMN^{(2)} == I^{(2)})\&\&(stMN^{(3)} == I^{(3)})\&\&(stMN^{(6)} == I^{(6)})] * \lambda$

$f_{5,6} = [((stMN^{(3)} == I^{(3)})\&\&(stMN^{(4)} == I^{(4)}))] * \lambda$

Tipo	Evento	Taxa	Tipo	Evento	Taxa
loc	tx_1,...tx_4	μ	sin	$g_{2,3}$	$f_{2,3}$
sin	tx_5	μ	sin	$g_{3,4}$	$f_{3,4}$
sin	$g_{1,2}$	$f_{1,2}$	sin	$g_{4,5}$	$f_{4,5}$
			sin	$g_{5,6}$	$f_{5,6}$

$partial\ reachability = (\text{nb I} == 6)$

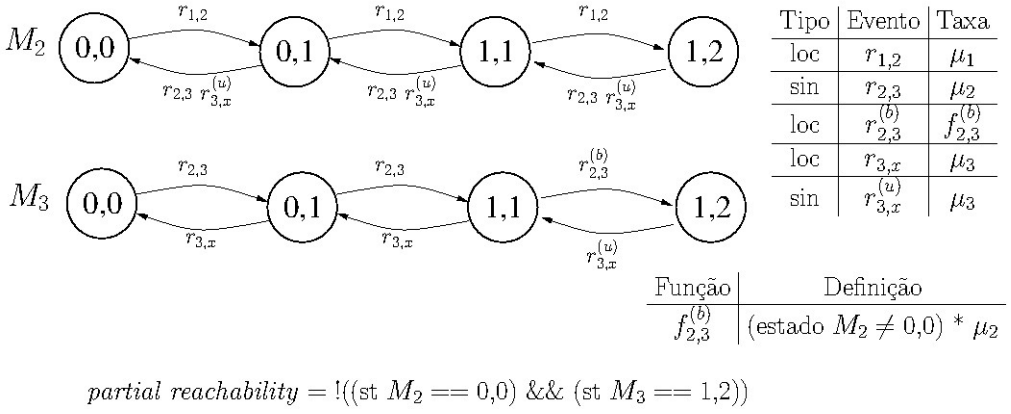
Figura 2.2 – Modelo adhoc com seis nodos [17]

A Figura 2.2 ilustra um modelo para cálculo de vazão líquida de dados de uma rede wireless adhoc com seis nodos. No modelo três diferentes tipos de autômatos foram definidos: *source*, *relay* e *sink*. O autômato *source* ($MN^{(1)}$) é responsável pela geração dos pacotes, ele possui dois estados: $I^{(1)}$ (ocioso) e $T^{(1)}$ (trasmitindo). Os automatos *relay* ($MN^{(i)}$, i = 2 .. N-1) possuem três estados:$I^{(1)}$ (ocioso), $R^{(1)}$ (recebendo) e $T^{(1)}$ (trasmitindo) e são responsáveis por passar adiante os pacotes gerados. O autômato *sink* ($MN^{(N)}$) é o último autômato da rede e possui dois estados: $I^{(N)}$ (ocioso) e $R^{(N)}$ (recebendo). Quando um nodo está no estado $T^{(i)}$ há um intervalo de interferência em três nodos antes e dois depois do que está transmitindo, isto é, nenhum outro nodo dentro deste intervalo pode transmitir simultaneamente. Maiores detalhes do modelo são encontrados em [17].

2.1.7 Linha de Produção

A Figura 2.3 apresenta o modelo SAN equivalente a um modelo de rede de filas de uma linha de produção com três estações, apresentado em [19]. Neste modelo SAN cada estação M_i e seu *buffer* B_i é modelada como um autômato, chamado M_i. A primeira estação M_1 não é modelada pois é considerado que esta estação nunca deixa de enviar trabalho para a estação M_2 com taxa μ_1.

O número de estados de cada autômato é dado pela combinação da ocupação do *buffer* (n_i) e o estado (s_i) para a estação *i*, dessa forma, considerando que a estação M_2 tem um *buffer* B_2=1 e a estação M_3 tem um *buffer* B_3=1, o número de estados de cada autômato é respectivamente: (0,0; 0,1; 1,1; 1,2) e (0,0; 0,1; 1,1; 1,2). Se s_i=0, a estação *i* está vazia e ociosa, ou seja, ela não está trabalhando. Se s_i=1, a estação *i* está ocupada e pode estar ou não trabalhando, dependendo se está bloqueada ou não. Se s_i=2, a estação *i* está ocupada e está bloqueando a estação anterior e pode ou não estar trabalhando, dependendo se está bloqueada ou não. Por exemplo, se s_3=2 a estação três está trabalhando e bloqueando a segunda estação.



Tipo	Evento	Taxa
loc	$r_{1,2}$	μ_1
sin	$r_{2,3}$	μ_2
loc	$r_{2,3}^{(b)}$	$f_{2,3}^{(b)}$
loc	$r_{3,x}$	μ_3
sin	$r_{3,x}^{(u)}$	μ_3

Função	Definição
$f_{2,3}^{(b)}$	(estado $M_2 \neq 0{,}0$) * μ_2

$$partial\ reachability = !((\text{st } M_2 == 0{,}0)\ \&\&\ (\text{st } M_3 == 1{,}2))$$

Figura 2.3 – Modelo SAN para Linha de Produção com três estações

O evento local $r_{1,2}$ com taxa μ_1 representa a chegada de um trabalho da estação M_1 para M_2. O evento local $r_{3,x}$ é a saída da estação M_3 enquanto que o evento sincronizante $r_{2,3}$ é a passagem de trabalho entre as estações M_2 e M_3. Os eventos $r_{2,3}^{(b)}$ e $r_{3,x}^{(u)}$ representam a relação entre as estações M_2 and M_3, ou seja, a estação M_3 está ocupada e bloqueando a estação M_2 ($r_{2,3}^{(b)}$) ou ela completa seu trabalho e desbloqueia M_2 ($r_{3,x}^{(u)}$). Maiores detalhes sobre os modelos podem ser encontrados em [19].

2.2 Verificação de Modelos

Verificação de Modelos, em inglês *Model Checking*, é uma técnica utilizada para verificar a correção de sistemas. Nesta abordagem o sistema a ser verificado é modelado como um sistema

de transição de estados finito e as especificações são expressas em lógica temporal [26], tema da próxima seção.

Conforme Clarke [10], o problema de *Model Checking* pode ser definido como: seja M um modelo do sistema, isto é, um grafo de transição de estados e f uma especificação descrita em lógica temporal, encontre todos os estados s de M tal que $M, s \models f$.

Através da exploração de todo o espaço estado do sistema de transição de estados é possível verificar se a fórmula é satisfeita. Em seu artigo em comemoração ao aniversário de *Model Checking* [10], Edmund Clarke lista algumas vantagens da técnica:

- Sem provas. O usuário do verificador não precisa construir provas. Em princípio, é necessário apenas entrar com a descrição do sistema a ser verificado e a especificação a ser avaliada. A verificação é automática.
- A lógica temporal pode facilmente especificar várias das propriedades que são necessárias quando se trata de sistemas concorrentes.
- Contraexemplos. Se uma especificação não é satisfeita, o *Model Checker* irá retornar um contraexemplo, ou seja, um caminho (sequências de estados e transições) que mostra a razão pela qual uma especificação não é satisfeita.

Também é possível a geração de uma testemunha para quando uma especificação é atendida, isto é, um caminho que ratifica a satisfação da fórmula pelo modelo.

A Figura 2.4 mostra um típico sistema de verificação de modelos (*Model Checker*) onde o verificador de modelos possui como entrada o modelo que será verificado e a especificação ou fórmula que será avaliada sobre o modelo. Como saída o verificador pode prover um contraexemplo ou uma testemunha, dependendo do resultado da verificação como falso ou verdadeiro ou, ainda, o modelo pode ser não tratável pelo verificador. Isso ocorre em virtude do problema de explosão do espaço de estados, comum em sistemas que possuem muitos componentes que interagem entre si.

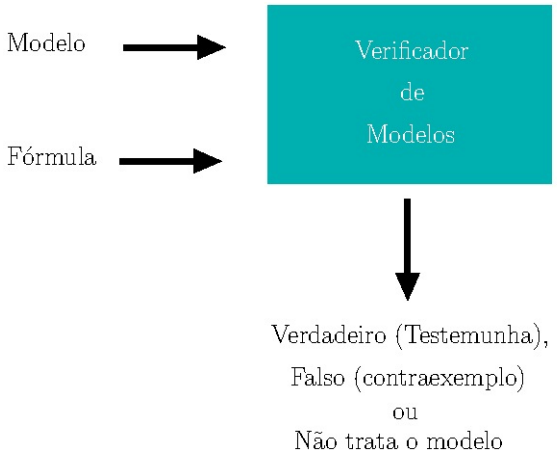


Figura 2.4 – Estrutura da Verificação de Modelos

2.3 Lógicas Temporais

As lógicas temporais são utilizadas para descrever a ordem relativa de eventos de um sistema. Elas podem descrever como o sistema se comporta ao longo do tempo ou que configurações ele toma ao longo do tempo de execução. Em *Model Checking*, as especificações descritas em lógica temporal são verificadas contra o modelo do sistema. Como dito na seção anterior, o modelo do sistema pode ser visto como um grafo de transição de estados. Cada estado no grafo é definido por um conjunto de proposições que são verdadeiras naquele estado. Desse modo, uma fórmula pode ser verdadeira para alguns estados e falsa para outros, numa mesma execução. Lamport [25] argumentou que as propriedades que podem ser impostas sobre os modelos encontram-se basicamente em duas categorias: *safety* e *liveness*.

- *Safety*: esse tipo de propriedade tipicamente é caracterizada como "algo ruim nunca acontece". Um sistema satisfaz tal propriedade se não realiza alguma tarefa proibida. Por exemplo, o verificador jamais proverá ao usuário uma testemunha quando o usuário solicitar um contraexemplo.
- *Liveness*: esse tipo de propriedade é caracterizada por "algo bom acontecerá". Para satisfazer essa propriedade o sistema realizará alguma atividade esperada. Por exemplo, o verificador proverá ao usuário uma testemunha, quando o usuário solicitar uma testemunha.

As lógicas temporais oferecem diferentes pontos de visão em relação à estrutura de tempo em que o sistema muda:

- *Linear*: a lógica temporal linear (LTL - *Linear Temporal Logic*) assume o tempo como uma sequência de execuções do sistema onde cada possível caminho de computação é considerado separadamente, levando em consideração apenas uma sequência de execução.
- *Ramificada*: a lógica temporal ramificada considera vários caminhos alternativos a partir de um ponto dado, e todos os caminhos de computação são considerados simultaneamente. Dentro do grupo de lógicas temporais ramificadas estão a CTL (*Computation Tree Logic*) e a CTL*, que é um superconjunto de LTL e CTL.

Não fazem parte do escopo deste projeto a especificação de fórmulas em LTL ou CTL*. Maiores detalhes sobre estas lógicas temporais são encontrados em [23].

A CTL possui os mesmos operadores temporais de LTL, isto é, *eventually* ($\Diamond$), *for all* ($\Box$), *next* ($\bigcirc$) e *until* ($\cup$), porém, acrescidos de dois quantificadores de caminho que consideram os possíveis caminhos ou computações a partir de um determinado estado. Os quantificadores de caminho são: o existencial (denotado por $\exists$) e o para todo ou universal (denotado por $\forall$). Por exemplo uma fórmula $\exists(true \cup \Phi)$ significa que existe uma computação ao longo da qual Φ é satisfeita. Existe no mínimo uma computação que satisfaz a fórmula.

Conforme Baier e Katoen [2], CTL tem uma sintaxe de dois estágios, classificados como fórmula de estado e fórmula de caminho. As fórmulas de estado são afirmações sobre os estados do modelo. Já as fórmulas de caminho expressam propriedades sobre ramos de computação.

Fórmulas CTL de estado são formadas de acordo com a seguinte gramática:

$$\varphi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

onde $a \in$ AP (conjunto de proposições atômicas) e φ é uma fórmula de caminho. Fórmulas CTL de caminho são formadas de acordo com a seguinte gramática:

$$\varphi ::= \bigcirc \Phi \mid \Phi_1 \cup \Phi_2$$

Os operadores temporais $\Diamond$ e $\Box$ não aparecem na gramática da CTL pois, conforme Baier e Katoen [2] eles podem ser derivados com o operador $\cup$, conforme:

$$\exists\Diamond\Phi = \exists(true \cup \Phi)$$
$$\forall\Diamond\Phi = \forall(true \cup \Phi)$$
$$\exists\Box\Phi = \neg\forall\Diamond\neg\Phi = \neg\forall(true \cup \neg\Phi)$$
$$\forall\Box\Phi = \neg\exists\Diamond\neg\Phi = \neg\exists(true \cup \neg\Phi)$$

A Figura 2.5 mostra algumas fórmulas CTL e seu significado em uma árvore de execução do sistema.

Para entendimento, considere um sistema de transição de estados $TS = (S, I, R, L)$ sobre um conjunto de proposições atômicas AP onde:

1. S representa o conjunto de estados do modelo.
2. I representa o conjunto de estados iniciais do modelo.
3. $R \subseteq S \times S$ é a relação de transição que deve ser total, ou seja, para todo estado $s \in S$ há um estado $s' \in S$ tal que $R(s, s')$.
4. $L : S \to 2^{AP}$ é a função de rotula cada estado com o conjunto de proposições atômicas que são verdadeiras naquele estado.

Desse modo, interpretamos as fórmulas apresentadas na Figura 2.5 da seguinte forma:

- $\exists \bigcirc preto$ - a partir de s existe algum s' com $R(s, s')$ tal que $TS, s' \models preto$ ou seja, algum próximo estado satisfaz $preto$;
- $\forall \bigcirc preto$ - todo estado s' tal que $R(s, s')$ há $TS, s' \models preto$ ou seja, todos próximos estados satisfazem $preto$;

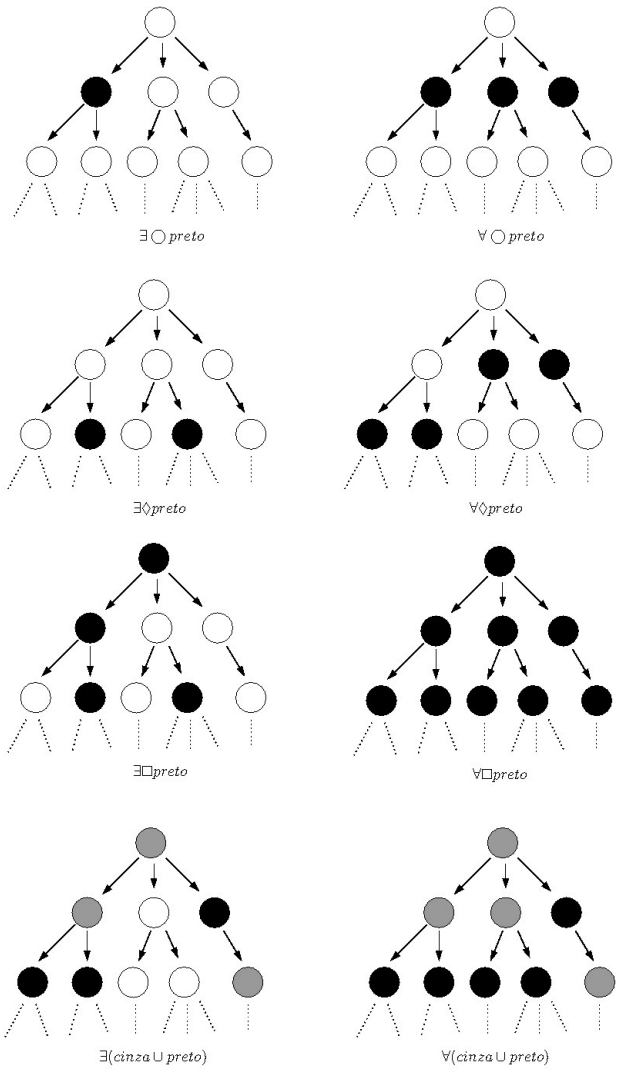


Figura 2.5 – Semântica de propriedades CTL.

- $\exists\Diamond preto$ - existe um caminho $R(s, s')$, $R(s', s'_1)$, $R(s'_1, s'_2)$, ..., onde há algum s ao longo do caminho tal que $TS, s \models preto$, ou seja, existe um caminho ao longo do qual existe um estado futuro que satisfaz $preto$;

- $\forall\Diamond preto$ - para todos os caminhos $R(s, s')$, $R(s', s'_1)$, $R(s'_1, s'_2)$, ..., há algum s ao longo de cada caminho tal que $TS, s \models preto$, ou seja, para todos os caminhos existe um estado futuro que satisfaz $preto$;

- $\exists\Box preto$ - existe um caminho $R(s, s')$, $R(s', s'_1)$, $R(s'_1, s'_2)$, ..., e para todos os s ao longo do caminho $TS, s \models preto$, ou seja, existe um caminho onde $preto$ é satisfeito ao longo de todos os estados do caminho. Note que $\Box$ inclui o estado inicial s;

- $\forall\Box preto$ - todos os caminhos $R(s, s')$, $R(s', s'_1)$, $R(s'_1, s'_2)$, ..., e para todos os s ao longo dos caminhos $TS, s \models preto$, ou seja, para todos os caminhos $preto$ é satisfeito ao longo de todos os estados dos caminhos. Note que $\Box$ inclui o estado inicial s;

- $\exists(cinza \cup preto)$ - existe um caminho $R(s, s')$, $R(s', s'_1)$, $R(s'_1, s'_2)$, ... que satisfaz $cinza \cup preto$, ou seja, há um s'_i ao longo do caminho tal que $TS, s'_i \models preto$ e para cada $j < i$ há $TS, s'_j \models cinza$, incluindo o estado inicial s, ou seja, existe um caminho ao longo do qual $cinza$ é satisfeito até que um estado $preto$ seja alcançado, desse modo o caminho satisfaz a fórmula $cinza \cup preto$;

- $\forall(cinza \cup preto)$ - todos os caminhos $R(s, s')$, $R(s', s'_1)$, $R(s'_1, s'_2)$, ... satisfazem $cinza \cup preto$, ou seja, há um s'_i ao longo do caminho tal que $TS, s'_i \models preto$ e para cada $j < i$ há $M, s'_j \models cinza$, incluindo o estado inicial s, ou seja, todos os caminhos satisfazem $cinza \cup preto$.

Os quantificadores de caminho podem ser representados por letras, a saber:

– o quantificador existencial $\exists$ pode ser representado pela letra *E*.

– o quantificador universal $\forall$ pode ser representado pela letra *A*.

Da mesma forma, os operadores temporais também podem ser representados nas fórmulas por letras, conforme os exemplos abaixo:

– $\exists \bigcirc preto$ ou $EXPreto$. Onde *X* representa o operador $\bigcirc$.

– $\exists \Diamond preto$ ou $EFPreto$. Onde *F* representa o operador $\Diamond$.

– $\exists \Box preto$ ou $EGPreto$. Onde *G* representa o operador $\Box$.

– $\exists(cinza \cup preto)$ ou $E(cinzaUpreto)$. Onde *U* representa o operador $\cup$.

- **CTL - Forma Existencial Normal**

A lei de dualidade [2] para quantificadores de caminho mostra que fórmulas quantificadas universalmente podem ser tratadas e representadas por fórmulas equivalentes quantificadas existencialmente. A Forma Existencial Normal (*Existential Normal Form - ENF*) não usa o quantificador universal e define as fórmulas em CTL usando os operadores básicos: $\exists\bigcirc$, $\exists\cup$ e $\exists\Box$.

Conforme Baier e Katoen [2], as fórmulas CTL em ENF são formadas de acordo com a seguinte gramática:

$$\varphi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists \bigcirc \Phi \mid \exists\Phi_1 \cup \Phi_2 | \exists\Box\Phi$$

Para o quantificador universal ser eliminado são necessárias transformações de acordo com as equivalências [2]:

$$\forall \bigcirc \Phi \equiv \neg\exists \bigcirc \neg\Phi$$
$$\forall\Phi \cup \Psi \equiv \neg\exists(\neg\Psi\cup(\neg\Phi \wedge \neg\Psi))\wedge\neg\exists\Box\neg\Psi$$
$$\forall\Diamond\Phi \equiv \neg\exists\Box\neg\Phi$$
$$\forall\Box\Phi \equiv \neg\exists\Diamond\neg\Phi = \neg\exists(true \cup \neg\Phi)$$

O algoritmo implementado no verificador de modelos SAN, apresentado neste trabalho, aceita como entrada uma fórmula CTL, que pode estar ou não na forma ENF, contudo, um *parser* é utilizado para converter a fórmula CTL de entrada para uma fórmula ENF equivalente. O fato de o algoritmo implementado computar somente fórmulas CTL em ENF reduz o escopo de fórmulas CTL que devem ser consideradas pela ferramenta. Esse fato não representa uma limitação em relação à capacidade de representação de uma fórmula em CTL, uma vez que, conforme Baier e Katoen [2], os demais operadores CTL podem ser derivados utilizando somente a forma ENF.

2.4 Contraexemplos e testemunhas

Uma das mais importantes vantagens de *Model Checking* sobre outras técnicas é a possibilidade da geração de contraexemplos para o caso de uma fórmula ser refutada [2, 12, 14]. Edmund Clarke [10] afirma que esta possibilidade tem valor inestimável para depuração de sistemas complexos e que muitas pessoas usam *Model Checking* em razão desta característica.

Quando um modelo sendo validado atende à uma especificação descrita em lógica temporal, o verificador de modelos gera uma resposta positiva, indicando ao usuário que o modelo atende à especificação. Neste caso, o verificador pode gerar, ainda, uma testemunha, isto é, uma execução que mostra que a especificação é atendida pelo modelo. Caso o resultado da verificação seja falso, ou seja, o modelo não atende à especificação avaliada, o verificador gera um contraexemplo, isto é, uma execução que mostra o porquê daquela especificação não ser atendida no modelo.

Uma questão importante e que deve ser levada em consideração, é a de que para CTL a geração de contraexemplos para fórmulas em ENF é mais complicada, devido ao quantificador existencial [2].

Para uma fórmula quantificada universalmente $\forall\varphi$, um contraexemplo seria uma sequência de estados que não satisfaçam φ. Já um contraexemplo para uma fórmula quantificada existencialmente ($\exists\varphi$), teria de garantir que não há execuções dentro do grafo de transição de estados do modelo que satisfazem φ. O contraexemplo para esta situação é todo o espaço de estados do modelo, o que torna inviável sua apresentação. O mesmo é verdade para a geração de testemunhas para fórmulas quantificadas universalmente, isto é, a testemunha teria de garantir que todas as execuções dentro do modelo satisfazem φ.

Desse modo, contraexemplos são gerados para fórmulas quantificadas universalmente, enquanto que testemunhas são geradas para fórmulas quantificadas existencialmente [2, 12]. A Tabela 2.1 mostra uma síntese das situações em que são gerados contraexemplos e testemunhas.

A seguir são apresentados os conceitos de contraexemplos e testemunhas para fórmulas CTL. Serão consideradas as fórmulas $\bigcirc\Phi$, $\Phi \cup \Psi$ e $\Box\Phi$, onde Φ e Ψ representam outras fórmulas CTL ou proposições atômicas aninhadas, pois, a partir destes operadores temporais outros podem ser derivados [2], conforme abordado pela Seção 2.3. Para entendimento considere o sistema de transição de estados $TS = (S, I, R, L)$ apresentado na mesma seção (2.3).

Tabela 2.1 – Geração de contraexemplos e testemunhas

Fórmula	Contraexemplo	Testemunha
$\forall\Box\Phi$	✓	
$\forall(\Phi_1 \cup \Phi_2)$	✓	
$\forall \bigcirc \Phi$	✓	
$\exists\Box\Phi$		✓
$\exists(\Phi_1 \cup \Phi_2)$		✓
$\exists \bigcirc \Phi$		✓

- **Operador Next** ($\bigcirc$)

  Contraexemplo: um contraexemplo para $\varphi = \bigcirc\Phi$ é um par de estados (*s, s'*) com $s \in I$ e $s' \in S$ tal que $s' \not\models \Phi$ e $R(s, s')$. A Figura 2.6 ilustra o contraexemplo para o operador *Next*, os círculos branco e cinza representam respectivamente os estados s e s'.

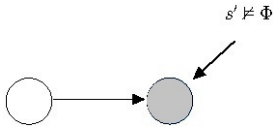


Figura 2.6 – Contraexemplo para o operador $\bigcirc$.

  Testemunha: uma testemunha para $\varphi = \bigcirc\Phi$ é um par de estados (*s, s'*) com $s \in I$ e $s' \in S$ tal que $s' \models \Phi$ e $R(s, s')$. A Figura 2.7 representa uma testemunha do operador *Next*, onde o estado s' representado pelo círculo cinza satisfaz Φ.

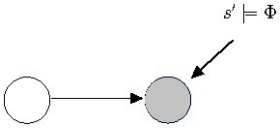


Figura 2.7 – Testemunha para o operador $\bigcirc$.

- **Operador Until** ($\cup$)

  Contraexemplo: um contraexemplo para $\varphi = \Phi \cup \Psi$ é um fragmento de caminho π que indica:

$$\pi \models \Box(\Phi \wedge \neg\Psi) \text{ ou}$$

$$\pi \models (\Phi \wedge \neg\Psi) \cup (\neg\Phi \wedge \neg\Psi)$$

  Para o primeiro caso, o contraexemplo é um caminho como ilustrado pela Figura 2.8. Note que no exemplo, todos os estados satisfazem $\Phi \wedge \neg\Psi$ e ao longo do caminho um ciclo é detectado, ou seja, o contraexemplo é um caminho que satisfaz $(\Phi \wedge \neg\Psi)$ com o formato:

$$s_0 s_1 ... s_{n-1} s_n s'_1 ... s'_r$$

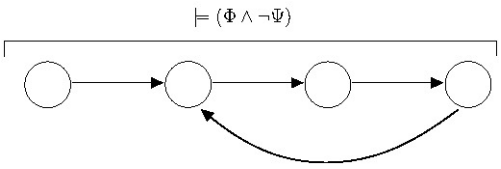


Figura 2.8 – Primeiro formato de contraexemplo para o operador $\cup$.

Onde $s_n = s'_r$, evidenciando um ciclo.

Para o segundo caso, o contraexemplo é um caminho que satisfaz $(\Phi \wedge \neg\Psi)$ com formato:

$$s_0 s_1 ... s_{n-1} s_n$$

Com $s_n \vDash (\neg\Phi \wedge \neg\Psi)$.

A Figura 2.9 ilustra o contraexemplo com este formato. O contraexemplo parte de um estado que satisfaz $\Phi \wedge \neg\Psi$ e percorre estados com a mesma característica até que um estado que satisfaça $\neg\Phi \wedge \neg\Psi$ seja alcançado.

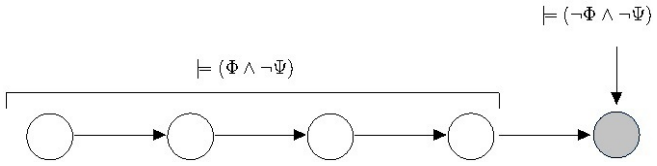


Figura 2.9 – Segundo formato de contraexemplo para o operador $\cup$.

Testemunha: uma testemunha para a fórmula $\varphi = \Phi \cup \Psi$ é um fragmento de caminho com $s_0 s_1 ... s_n$ onde:

$$s_n \models \Psi \text{ e } s_i \models \Phi \text{ para } 0 \leqslant i < n$$

As testemunhas podem ser determinadas por uma busca para trás iniciando no conjunto de estados Ψ [2].

A Figura 2.10 ilustra uma testemunha para o operador $\cup$. A testemunha parte de um estado que satisfaz Φ, percorrendo estados com a mesma característica até que um estado que satisfaça Ψ seja alcançado.

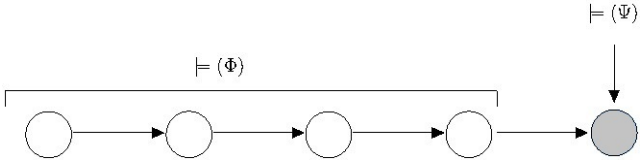


Figura 2.10 – Testemunha para o operador $\cup$.

- **Operador Always** ($\Box$)

  Contraexemplo: um contraexemplo para $\varphi = \Box\Phi$ é um fragmento de caminho $s_0 s_1 ... s_n$ tal que $s_i \models \Phi$ para $0 \leqslant i < n$ e $s_n \not\models \Phi$. Contraexemplos podem ser determinados por uma busca para trás, iniciando nos estados $\neg\Phi$. A Figura 2.11 ilustra um contraexemplo para o operador $\Box$. O contraexemplo é um caminho com estados que satisfazem Φ terminando com um estado que não satisfaz Φ.

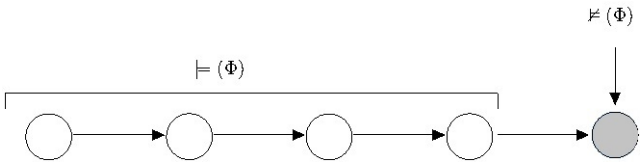


Figura 2.11 – Contraexemplo para o operador $\Box$.

  Testemunha: uma testemunha para mesma fórmula consiste num caminho que satisfaz Φ no formato:

$$s_0 s_1 ... s_{n-1} s_n s'_1 ... s'_r$$

  Com $s_n = s'_r$, evidenciando assim um ciclo.

  As testemunhas podem ser obtidas através da busca por um ciclo simples no dígrafo *G=(S,E)* onde o conjunto de arestas *E* é obtido das transições que emanam dos estados Φ, isto é, $E = (s, s')|R(s, s') \wedge s \models \Phi$. A Figura 2.12 ilustra uma testemunha para este caso, onde um ciclo é detectado no segundo e quarto estados.

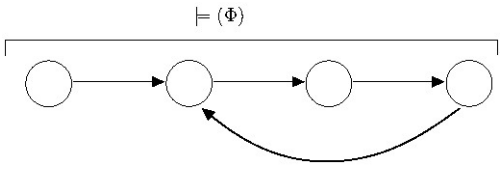


Figura 2.12 – Testemunha para o operador $\Box$.

Como exemplo considere a Figura 2.13, que representa o grafo de transição de estados do modelo do Jantar dos Filósofos, apresentado pela Figura 2.1. Nesta figura, a letra *T* representa o estado *Thinking*, a letra *R*, o estado *Right* e a letra *L* o estado *Left*. Os filósofos aparecem na seguinte ordem: Fil2, Fil1 e Fil0.

Agora, considere as seguintes fórmulas CTL: EFEG(estado do filósofo um = *Right*) e EF(estado do filósofo zero = *Left*). A primeira fórmula afirma que existe um caminho no grafo de execução que leva a um ciclo (representado pelo operador EG) onde o filósoso 1 permanece no estado *Right*. A Figura 2.14 mostra a testemunha para esta fórmula. Já a segunda fórmula afirma que existe no futuro um caminho que levará a um estado onde o filósofo zero está comendo. Uma testemunha para esta fórmula é apresentada pela Figura 2.15.

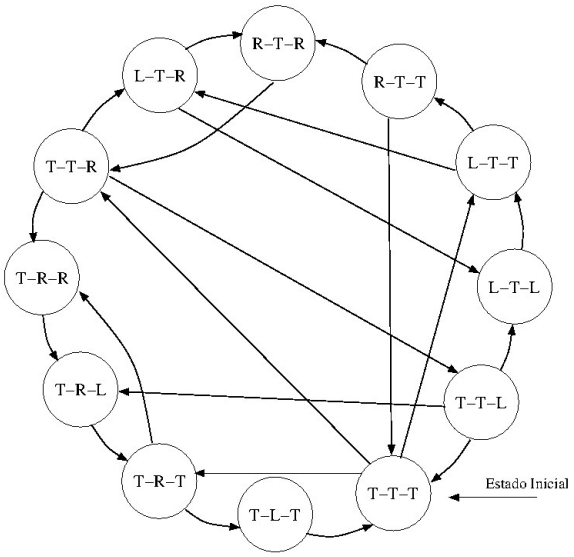


Figura 2.13 – Grafo de transição de estados para o modelo da Figura 2.1

.

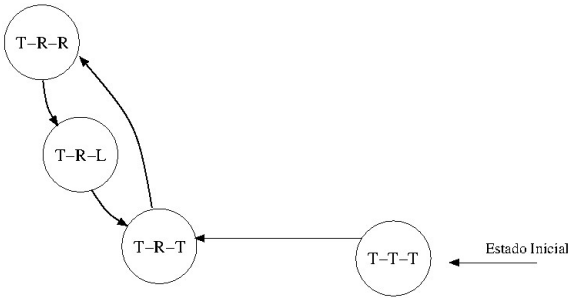


Figura 2.14 – Testemunha para fórmula EFEG(estado do filósofo um = *Right*)

.

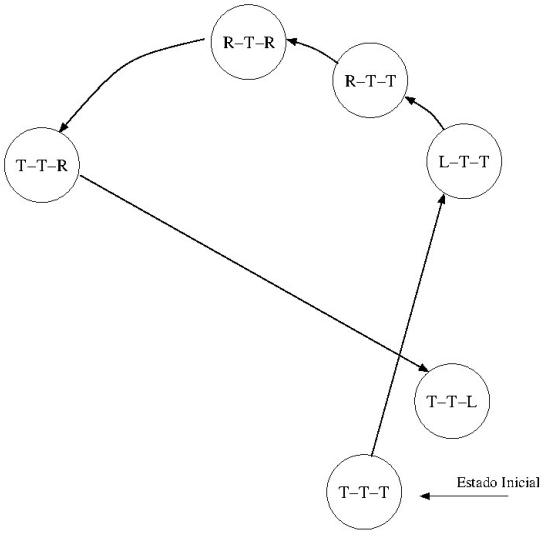


Figura 2.15 – EF(estado do filósofo zero = *Left*)

.

- **Contraexemplos e Testemunhas em árvore**

Os conceitos apresentados acima tratam de contraexemplos e testemunhas lineares ou *trace* contraexemplos e testemunhas. A maioria dos verificadores de modelos, incluindo o NuSMV [9], geram resultados em formato de *trace* para especificações CTL [14].

Contudo, considere o exemplo $\forall\Box\neg x \vee \forall\Diamond\neg y$ retirado de [13]. Um contraexemplo para esta fórmula tem de descrever dois *traces*, um finito que leve a um estado satisfazendo *x* e outro em que há um caminho infinito ao longo do qual *y* é sempre verdade. Isto significa que o contraexemplo tem de ser um modelo para $\exists\Diamond x \wedge \exists\Box\ y$. Outro exemplo mais complicado é dado por fórmulas como $\forall\Diamond\forall\bigcirc x$ onde um contraexemplo tem de provar que existe um caminho ao longo do qual um estado $\neg x$ pode ser alcançado em um passo.

Os exemplos acima demostram que *trace* contraexemplos não são completos para fórmulas quantificadas universalmente [7, 10, 13, 14].

Clarke e Veith [14] afirmam que a adequação de *trace* contraexemplos foi um ponto cego por anos, mas que começou a ser considerado por [7, 13]. Ainda conforme Clarke e Veith [14], *trace* contraexemplos estão intimamente relacionados ao fragmento linear da CTL, ou seja, CTL $\cap$ LTL.

A geração de contraexemplos e testemunhas no formato de árvore não faz parte do escopo deste trabalho, portanto, a ferramenta irá gerar como resultado apenas contraexemplos e testemunhas lineares.

- **Contraexemplos para fórmulas na Forma Existencial Normal - ENF**

Conforme citado na Seção 2.3, o algoritmo implementado na ferramenta apresentada neste trabalho considera apenas fórmulas CTL na Forma Existencial Normal (ENF). Isso delimita o escopo da ferramenta a buscas por testemunhas dentro do modelo, visto que apenas testemunhas podem ser obtidas a partir de fórmulas quantificadas existencialmente.

Dessa forma, é necessário fazer a manipulação da árvore sintática que representa a fórmula ENF equivalente à fórmula quantificada universalmente, para que a testemunha que deverá ser encontrada equivalha a um contraexemplo da fórmula original, quantificada universalmente. Para tanto, é realizada a negação da fórmula em ENF que é equivalente a fórmula original, chegando aos resultados apresentados na Tabela 2.2. A coluna *Fórmula original* apresenta a fórmula quantificada universalmente. A coluna *ENF equivalente* mostra a fórmula em ENF equivalente à fórmula original. Já a coluna *Contraexemplo* mostra o formato que a testemunha deve ter para representar um contraexemplo da fórmula original.

Tabela 2.2 – Contraexemplos para fórmulas em ENF

Fórmula original	**ENF equivalente**	**Contraexemplo**
$\forall\Box\Phi$	$\neg\exists(true \cup \neg\Phi)$	$\exists(true \cup \neg\Phi)$
$\forall(\Phi_1 \cup \Phi_2)$	$\neg\exists(\neg\Phi_2 \cup (\neg\Phi_1 \wedge \neg\Phi_2)) \wedge \neg\exists\Box\neg\Phi_2$	$\exists(\neg\Phi_2 \cup (\neg\Phi_1 \wedge \neg\Phi_2)) \vee \exists\Box\neg\Phi_2$
$\forall\bigcirc\Phi$	$\neg\exists\bigcirc\neg\Phi$	$\exists\bigcirc\neg\Phi$

2.5 Trabalhos Relacionados

No trabalho intitulado *Implementing a CTL Model Checker* [21], Heljanko apresenta um algoritmo não recursivo para verificação de modelos. A principal contribuição do autor no trabalho, foi a criação de um algoritmo para *model checking* com complexidade de tempo e uso de memória iguais ou menores do que os algoritmos ALMC (*The Local Model Checking Algorithm*) [33] e EMC (*Extended Model Checking*) [18], utilizados como base para o desenvolvimento.

A estratégia utilizada pelo autor foi a de armazenar os valores verdade de cada subfórmula para cada estado do modelo. Para tanto, foi definida uma ordem de avaliação para fórmulas CTL. Conforme o exemplo dado pelo autor, a ordem parcial para a fórmula $\forall$ *(p* $\cup\exists$ *(q* $\cup$ $\neg$*p))* é a mostrada pela figura 2.16. As setas indicam a dependência entre as subfórmulas e os valores ao lado, a ordem total entre elas.

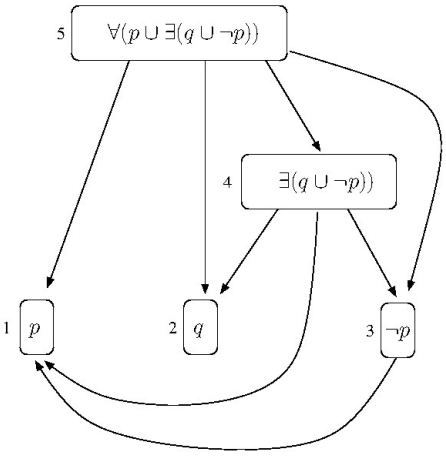


Figura 2.16 – Ordem parcial para fórmula $\forall$ (p $\cup$ $\exists$ (q $\cup$ $\neg$p)). Fonte: [21].

O algoritmo armazena em uma matriz L[s][i], onde *s* é um estado que pertence ao conjunto de estados *S* do modelo e *i* é o índice de uma subfórmula que compõe a fórmula de nível maior, o valor verdade de cada subfórmula para cada estado. Conforme Heljanko [21], o algoritmo tem uma estrutura similar aos algoritmos ALMC e EMC diferenciando-se pelo fato de possuir o recurso de contraexemplos e testemunhas e pela maneira como faz a verificação do operador *until*. Baseado na definição deste operador, o algoritmo marca ou não estados, dependendo do valor verdade da subfórmula da esquerda e da direita em cada estado.

A Tabela 2.3 mostra como é realizada a rotulação dos estados para o caso da fórmula $\Phi_1 \cup \Phi_2$. A coluna *Rótulo para fórmula* representa o valor verdade para a fórmula $\Phi_1 \cup \Phi_2$ em um determinado estado. A coluna *Rótulo lado esquerdo* representa o valor verdade de Φ_1 para o estado, a coluna *Rótulo lado direito* o valor verdade de Φ_2, enquanto que a coluna *Marcado* indica se o algoritmo já definiu o valor verdade da fórmula de nível maior para aquele estado. Como podemos observar, existe apenas um caso em que o algoritmo não pode determinar o valor verdade da fórmula. Este caso é aquele em que o lado esquerdo da fórmula com operador *until* é verdade

e seu lado direito é falso. Para este caso, o algoritmo precisa avaliar um estado posterior a fim de determinar o valor verdade da fórmula.

Tabela 2.3 – Avaliação do operador *until*

Rótulo lado esquerdo	**Rótulo lado direito**	**Marcado**	**Rótulo para fórmula**
Falso	Falso	Verdadeiro	Falso
Falso	Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso	Falso
Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro

Tendo o espaço de estados rotulado, a construção de uma testemunha para a fórmula $\exists(\Phi_1 \cup \Phi_2)$ torna-se simples. O algoritmo realiza uma busca em profundidade no modelo, caminhando por estados que satisfaçam Φ_1 até que um estado que satisfaça Φ_2 seja encontrado. À medida que o caminhamento é realizado, os estados percorridos são armazenados em uma lista, que ao final conterá a testemunha para a fórmula.

Em *Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking* [11], Clarke *et. al* apresentam os algoritmos simbólicos utilizados no NuSMV [9] responsáveis por computar a satisfação de fórmulas e construir os contraexemplos e testemunhas lineares da ferramenta. Os algoritmos apresentados no trabalho constroem contraexemplos e testemunhas respeitando um conjunto *H* de *fairness constraints*, definido por um conjunto de fórmulas CTL. Os autores também fazem considerações sobre a construção de *traces* sob *fairness constraints* com o menor tamanho possível. Contudo, segundo eles, este problema é *NP-complete*. Mesmo assim, os algoritmos implementados fazem uso de técnicas que garantem um *trace* que seja o mais curto possível.

Por fim, o artigo apresenta considerações sobre a construção de contraexemplos e testemunhas para fórmulas CTL*, pois, segundo os autores, muitas propriedades sobre projetos de protocolos e circuitos não podem ser expressadas pela CTL, nesses casos, faz-se o uso da CTL extendida, isto é, CTL*.

O SAN *Model Checker*, assim como o NuSMV, realiza a verificação simbólica de fórmulas CTL, o verificador faz uso de diagramas de decisão multivalorados (*Multivalued Desicion Diagrams* - MDD, em inglês) para codificar o espaço de estados atingível e os estados que satisfazem a fórmula avaliada. Recomenda-se a leitura dos trabalhos de Sales e Plateau [31] e Sales [29] para maiores detalhes sobre o funcionamento dos diagramas de decisão multivalorados. Diferentemente do NuSMV, que também utiliza algoritmos simbólicos para a geração de contraexemplos, o SAN *Model Checker* utiliza algoritmos de busca em profundidade no modelo, empregando uma estratégia semelhente à aplicada por Heljanko [21] para realizar a geração de contraexemplos e testemunhas como saída para o usuário. A estratégia utilizada para a geração dos resultados é apresentada no Capítulo 4.

3. SAN MODEL CHECKER

Este capítulo descreve a arquitetura do Verificador de Modelos Descritos em Redes de Autômatos Estocásticos (*SAN Model Checker*). Cabe ressaltar que o conteúdo apresentado neste capítulo é uma contribuição conjunta entre o autor desta dissertação e os alunos de mestrado Eli Maruani e Lucas Oleksinski. Os primeiros resultados alcançados na implementação da ferramenta foram publicados em [15] e [24].

A arquitetura apresentada pela Figura 3.1 é dessa forma, base para o desenvolvimento da arquitetura que será apresentada no Capítulo 4, onde alterações foram feitas com intenção de agregar à ferramenta o recurso de contraexemplos e testemunhas. A implementação do Verificador é baseada no algoritmo de satisfação de fórmulas CTL (*Computation Tree Logic*) em ENF (*Existential Normal Form*) apresentado por Baier e Katoen [2].

Diagramas de Decisão Multivalorados (*Multivalued Decision Diagrams - MDD*) são utilizados para a codificação do espaço de estados atingível (*Reachable Space State - RSS*). MDD's são estruturas de dados representadas por um grafo acíclico direcionado utilizados para armazenar informações de maneira eficiente e sem redundâncias [31].

A Figura 3.1 mostra a arquitetura do verificador implementado.

- Modelo SAN - Modelo SAN que será verificado.
- Estado Inicial - um ou mais estados iniciais devem ser definidos na função de *partial reachability*. Esta função é o ponto de partida para a geração do RSS.
- Proposições Atômicas - proposições atômicas são afirmações sobre o modelo que são avaliadas como verdadeiras ou falsas para cada estado global atingível no modelo [12]. Uma expressão atômica para SAN é uma expressão SAN sem operadores temporais.
- Propriedade CTL - qualquer propriedade em CTL [23] onde as proposições atômicas são expressões de SAN.
- Compilador - o compilador gera o descritor markoviano, isto é, um conjunto de tensores que quando operados por álgebra tensorial permitem a obtenção de todos os estados da Cadeia de Markov. As operações soma tensorial e produto tensorial reproduzem a interação e como os diferentes autômatos sincronizam os eventos [16].
- Geração do Espaço de Estados Atingível (RSS) - o descrito Markoviano é utilizado como um sistema de transição de estados. Dado um estado inicial, o gerador RSS verifica nos tensores, se há um evento que possa ser disparado. Caso isso seja possível, um sucessor é alcançado, computado junto ao RSS e codificado via MDD. Esta é a mesma técnica empregada no SAN Lite-Solver [30].
- Função de Rotulação - esta função rotula todos os estados com todas as proposições atômicas que são verdadeiras naquele estado. Dado o conjunto de proposições atômicas e o espaço

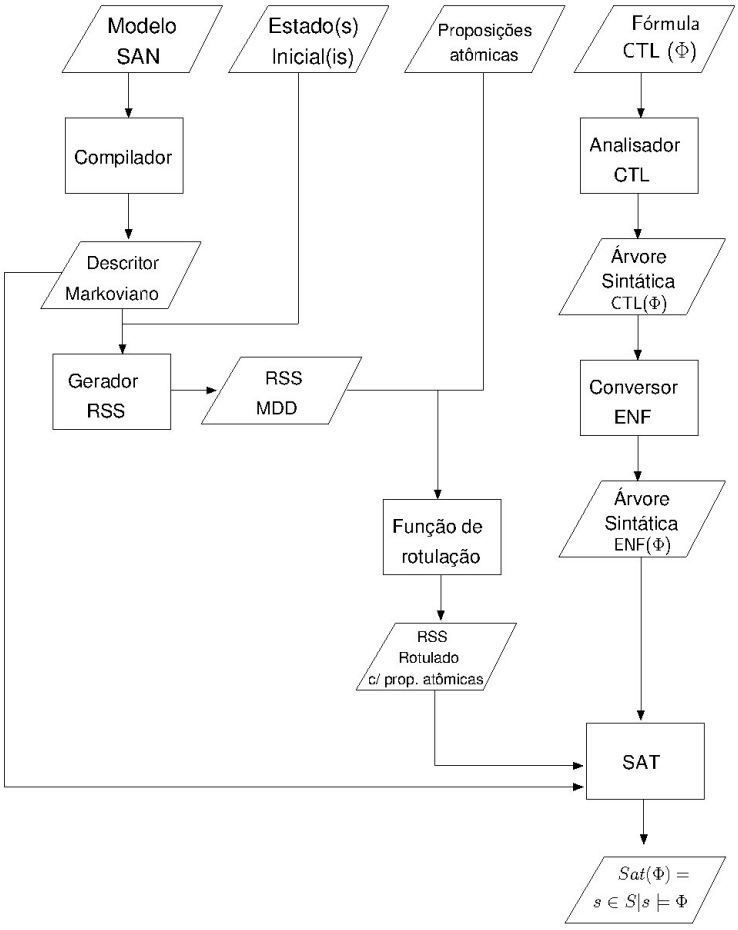


Figura 3.1 – Arquitetura *SAN Model Checker* [15]

.

de estados atingível, a função cria para cada proposição atômica um MDD que codifica os estados em que aquela proposição é verdade.

- Analisador CTL - este módulo manipula uma fórmula CTL e a armazena numa árvore sintática.
- Conversor ENF - o objetivo do módulo de conversão para ENF é criar uma árvore sintática da fórmula em ENF. Para tanto ele manipula a árvore sintática da CTL gerando uma árvore sintática em ENF usando regras de equivalência descritas em [2].
- Algoritmo de Satisfação (SAT) - este algoritmo realiza a verificação da fórmula de entrada no modelo. A implementação é baseada no algoritmo de Baier e Katoen [2] apresentado pelo Algoritmo 1.
- $Sat(\Phi) = \{s \in S | s \models \Phi\}$ - conjunto de estados que satisfazem Φ retornado pelo SAT.

Note que o algoritmo de satisfação de fórmulas considera apenas os operadores básicos de caminho de fórmulas CTL em ENF, ou seja, $\exists\bigcirc$, $\exists\cup$ e $\exists\Box$. O operador $\exists\Diamond$ não é considerado, pois, uma fórmula no formato $\exists\Diamond\Phi$ pode ser convertida para uma fórmula equivalente com operador $\exists\cup$, $\exists(true \cup \Phi)$.

Algoritmo 1: Satisfação de fórmulas CTL em ENF - Sat(Φ) [2]

Entrada: sistema de transição finito *TS* com conjunto de estados S e fórmula Φ em ENF
Saída: $Sat(\Phi) = \{\ s \in S \mid s \models \Phi\ \}$
/* computação recursiva dos conjuntos $Sat(\Psi)$ para todas sub-fórmulas Ψ de Φ */

```
1  início
2    caso Φ
3      selecione true faça
4        retorna S;
5      fim selec
6      selecione a faça
7        retorna { s ∈ S | a ∈ L(s) };
8      fim selec
9      selecione Φ1 ∧ Φ2 faça
10       retorna Sat(Φ1) ∩ Sat(Φ2);
11     fim selec
12     selecione ¬Ψ faça
13       retorna S \ Sat(Ψ);
14     fim selec
15     selecione ∃ ◯ Ψ faça
16       retorna { s ∈ S | Post(s) ∩ Sat(Ψ) ≠ ∅ };
17     fim selec
18     selecione ∃(Φ1 ∪ Φ2) faça
         /* Computa o menor ponto-fixo */
19       T := Sat(Φ2);
20       enquanto {s ∈ Sat(Φ1) \ T | Post(s) ∩ T ≠ ∅} ≠ ∅ faça
21         seja s ∈ { s ∈ Sat(Φ1) \ T | Post(s) ∩ T ≠ ∅ };
22         T := T ∪ { s };
23       fim enqto
24       retorna T;
25     fim selec
26     selecione ∃□Ψ faça
         /* Computa o maior ponto-fixo */
27       T := Sat(Ψ);
28       enquanto {s ∈ T | Post(s) ∩ T = ∅} ≠ ∅ faça
29         seja s ∈ { s ∈ T | Post(s) ∩ T = ∅ };
30         T := T \ { s };
31       fim enqto
32       retorna T;
33     fim selec
34   fim selec
35 fim
36 ---
```

37 S: Espaço de estados atingíveis (RSS)
38 $Post(s)$: Estados posteriores (sucessores) de um estado s
39 L: $S \to 2^{AP}$: associa a cada $s \in S$ o conjunto de proposições atômicas $L(s)$ satisfeitas em s
40 ---

O algoritmo possui como entrada um sistema finito de transição de estados *TS* com um conjunto de estados *S* e a fórmula Φ que será verificada. A saída do algoritmo varia de acordo com o tipo de fórmula de entrada:

- $true$ - retorna o conjunto de estados atingível do modelo (RSS).
- a - retorna o conjunto de estados onde a proposição a é verdadeira.
- $(\Phi_1 \wedge \Phi_2)$ - retorna a interseção entre os conjuntos de estados que satisfazem Φ_1 e Φ_2.
- $\neg\Psi$ - retorna a diferença entre o conjunto de estados atingível *S* e o conjunto de estados que satisfaz Ψ.
- $\exists \bigcirc \Psi$ - retorna o conjunto de estados cujo posteriores satisfazem Ψ.
- $\exists(\Phi_1 \cup \Phi_2)$ - inicialmente calcula o menor ponto-fixo, isto é, o conjunto *T* de estados que satisfazem Φ_2 para depois adicionar a este conjunto os estados que satisfazem Φ_1 e que possuem posteriores que satisfaçam Φ_2, para então retornar o conjunto *T*.
- $\exists\Box\Psi$ - inicialmente calcula o maior ponto-fixo, isto é, o conjunto *T* de estados que satisfazem Ψ para depois remover deste conjunto aqueles estados que não possuem posteriores que satisfaçam Ψ, para então retornar o conjunto *T*.

4. GERAÇÃO DE CONTRAEXEMPLOS E TESTEMUNHAS

O algoritmo de satisfação de fórmulas implementado no SAN *Model Checker* trata apenas fórmulas CTL em ENF. Desse modo, ele recebe como entrada uma especificação f' já devidamente tratada pelo *Conversor ENF* que é semanticamente equivalente a especificação f dada como entrada no Verificador (ver Capítulo anterior). Essa característica delimita o escopo da ferramenta a buscas por testemunhas para as especificações avaliadas no modelo.

Destaca-se que, para a geração de um contraexemplo para uma especificação, se faz necessário tratá-la, antes de enviá-la para o algoritmo de satisfação de fórmulas, de forma que a testemunha que será gerada equivalha a um contraexemplo da fórmula original dada como entrada na ferramenta. Para tanto, quando o usuário faz a solicitação de um contraexemplo, a especificação f' tratada pelo *Conversor ENF* é negada gerando uma nova especificação f'' que é equivalente a um contraexemplo para a especificação original f (ver Tabela 2.2 da Seção 2.4 para alguns exemplos deste tratamento).

A estratégia adotada na geração de contraexemplos e testemunhas foi norteada pelo trabalho de Keijo Heljanko [21], brevemente apresentado na Seção 2.5 desta dissertação, que utiliza o espaço de estados rotulado com o valor verdade de cada (sub)fórmula para o algoritmo de cálculo de testemunhas. Essa estratégia simplifica a busca por uma testemunha, pois, o cálculo dela resume-se a caminhar linearmente a partir de um estado inicial válido escolhendo o próximo estado da testemunha conforme o rótulo do estado atual.

O SAN *Model Checker* limita-se à geração de contraexemplos e testemunhas lineares, ou seja, *traces* de execução do modelo, de acordo com os conceitos de contraexemplos e testemunhas para os operadores temporais CTL apresentados por Baier e Katoen [2]. Sendo assim, resultados em formato de árvore (ver *Contraexemplos e Testemunhas em árvore* da Seção 2.4) não são gerados pelo verificador. Quando a ferramenta identifica um resultado em formato de árvore, ela apenas mostra o estado a partir do qual o resultado torna-se ramificado, marcando a (sub)fórmula válida naquele estado.

Para agregar ao verificador o recurso de geração de contraexemplos e testemunhas foram realizadas algumas modificações na arquitetura da ferramenta apresentada pela Figura 3.1 do Capítulo 3. A Figura 4.1 apresenta a arquitetura construída para permitir a geração de contraexemplos e testemunhas. Na figura, os itens com linha pontilhada representam as alterações feitas (contribuição deste trabalho) na arquitetura da ferramenta que possibilitaram a geração de contraexemplos e testemunhas como saída para o usuário.

As modificações realizadas acresceram à arquitetura os itens descritos a seguir:

- Contraexemplo ou testemunha - opção feita pelo usuário para o resultado gerado pela ferramenta ao verificar uma propriedade. Ele poderá optar pela geração de um contraexemplo ou uma testemunha.

- Tratador ENF - este método pode modificar a fórmula ENF de acordo com a saída solicitada pelo usuário. Caso o usuário opte por um contraexemplo, este método negará a fórmula ENF antes de enviá-la para o SAT. Caso o usuário opte por uma testemunha, nenhuma alteração é feita na fórmula ENF.

- RSS Rotulado com (sub)fórmulas - ao término do algoritmo de satisfação (SAT), os índices que codificam no MDD os estados que satisfazem a fórmula CTL e as subfórmulas que a compõem são armazenados na lista *Labels* (ver Algoritmo 2).

- Verificador de estado inicial - este método realiza a interseção entre o conjunto de estados iniciais do modelo e o conjunto de estados que satisfazem a fórmula Φ, desse modo se há algum estado inicial que satisfaça a fórmula este é armazenado na lista *trace* e torna-se o ponto de partida para construção da testemunha. Caso contrário, ou seja, caso não haja algum estado inicial do modelo que satisfaça a fórmula, o programa é encerrado e uma mensagem é mostrada ao usuário. O Algoritmo 3 mostra o pseudo-código desta rotina.

- Estado inicial (s) - um dos estados iniciais encontrados pelo *Verificador de estado inicial* que satisfaz a fórmula Φ e que está armazenado na lista *trace*.

- Gerador de testemunha - método que monta, a partir de um estado inicial que satisfaz a fórmula Φ, o *trace* que representa a testemunha para a fórmula. Testemunha que pode ter valor semântico de um contraexemplo. Detalhes em *Contraexemplos para Fórmulas na Forma Existencial Normal - ENF* da Seção 2.4. O Algoritmo 4 mostra o pseudo-código desta rotina.

- Testemunha - *trace* gerado como saída para o usuário.

Além das alterações na arquitetura, modificações foram feitas na função de rotulação de estados. Inicialmente ela rotulava apenas as proposições atômicas utilizadas na fórmula CTL que é avaliada. Contudo, como o formalismo SAN não possui uma descrição explícita do(s) estado(s) inicial(is) do modelo, a rotina passou a rotular também os estados que satisfazem a *partial reachability* dos modelos. Da perspectiva deste trabalho, estes passaram a ser considerados os estados iniciais.

A geração do *trace*, responsabilidade do *Gerador de Testemunha*, Algoritmo 4, é realizada após a execução do algoritmo de satisfação de fórmulas CTL em ENF, SAT. O *Gerador de Testemunha* possui como entrada a fórmula ENF Φ da qual o resultado que será gerado deve ser testemunha, a lista Labels que armazena o espaço de estados rotulado com as subfórmulas de Φ, o sistema de transição de estados TS e a lista *trace*, que já contém o estado inicial que satisfaz a fórmula (Φ) e que ao final da execução do *Gerador de Testemunha* armazenará a testemunha encontrada. São considerados os mesmos operadores de caminho tratados pelo Algoritmo 1, isto é, $\exists\bigcirc$, $\exists\Box$ e $\exists\cup$, além de *not*($\neg$) e a conjunção *and* ($\wedge$).

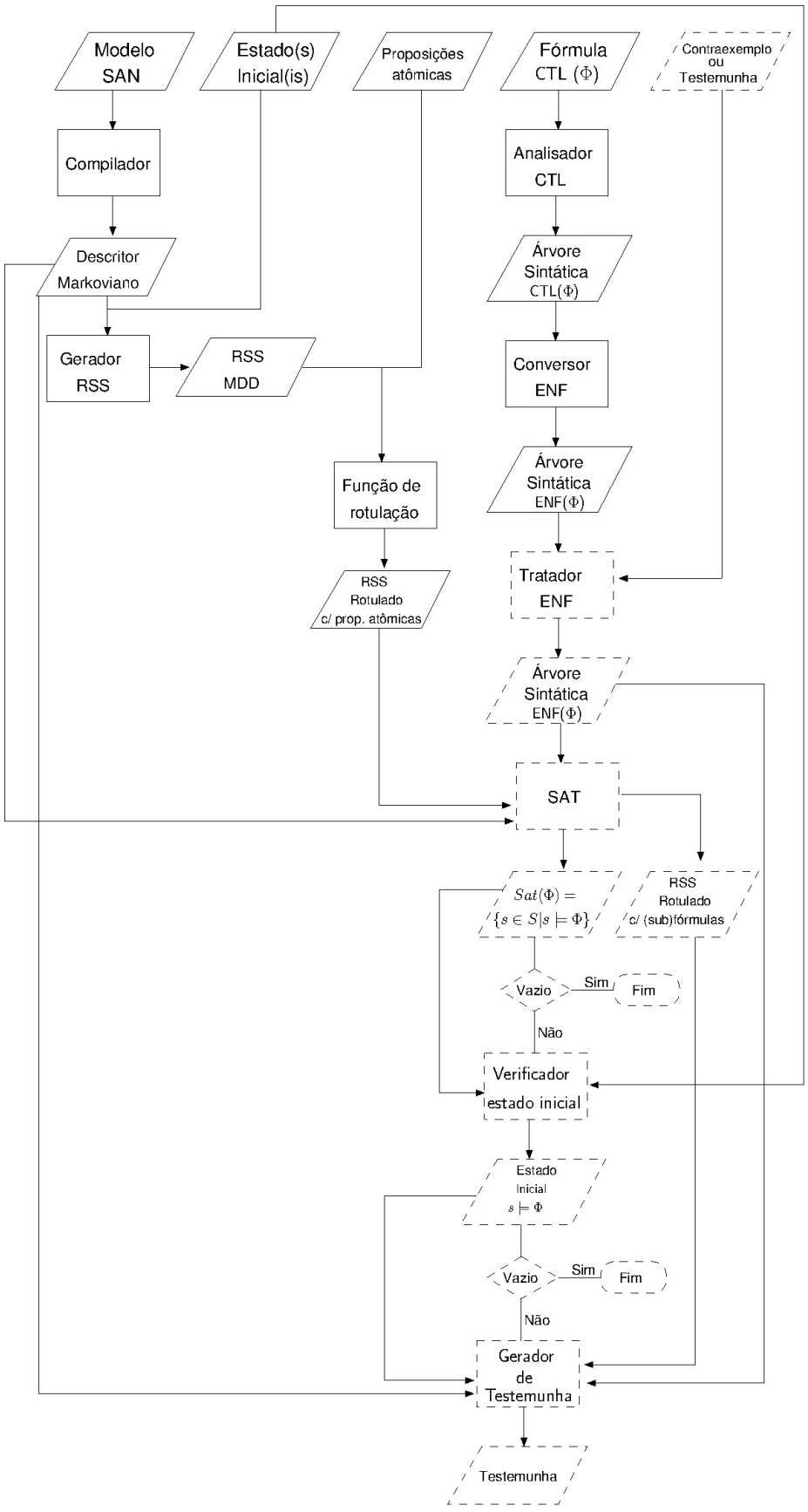


Figura 4.1 – Arquitetura *SAN Model Checker* com Contraexemplos e Testemunhas

.

Para empregar uma abordagem semelhante a usada por Heljanko [21], o algoritmo de satisfação de fórmulas CTL em ENF apresentado pelo Algoritmo 1 foi alterado, dando origem ao Algoritmo 2. A alteração realizada trata da inclusão da lista *Labels* no algoritmo. Esta lista armazena em ordem os valores que codificam no MDD os estados que satisfazem cada subfórmula que compõe a fórmula CTL avaliada, confome descrito abaixo:

- a - neste caso a lista Labels armazena o valor que codifica os estados rotulados com a proposição atômica a (linha 7);
- $\Phi_1 \wedge \Phi_2$: calcula-se recursivamente $Sat(\Phi_1)$ e $Sat(\Phi_2)$ que adicionarão estes rótulos à lista Labels e logo após, na linha 11, acontece a interseção entre o conjunto de estados que satisfaz Φ_1 e o conjunto de estados que satisfaz Φ_2. O valor que codifica os estados resultantes desta interseção também é armazenado em Labels, na linha 12;
- $\neg\Psi$: para a fórmulas negadas, a lista Labels armazena os valores que codificam os estados que satisfazem a diferença entre o espaço de estados S e o conjunto de estados que satisfaz Ψ. Esta diferença é calculada pelo SAT na linha 16 e o valor que codifica os estados resultantes desta diferença são armazenados em Labels, linha 17;
- $\exists \bigcirc \Psi$: para fórmulas com o operador temporal $\bigcirc$, a lista Labels armazena o valor que codifica os estados que satisfazem $\exists \bigcirc \Psi$, linha 22, este conjunto de estados é calculado pelo algoritmo SAT na linha anterior;
- $\exists (\Phi_1 \cup \Phi_2)$: calcula-se recursivamente $Sat(\Phi_2)$ (linha 26) e $Sat(\Phi_1)$ (linha 27), adicionando estes rótulos à lista. Logo após, o SAT calcula o conjunto de estados que satisfaz $\Phi_1 \cup \Phi_2$ adicionando à lista o valor que codifica os estados que estão nesse conjunto, linha 31.
- $\exists\Box\Psi$: para este caso a lista Labels armazena o valor que codifica os estados que satisfazem $\exists\Box\Psi$, linha 40.

Dessa forma, a lista *Labels*, ao final do algoritmo de satisfação, guarda o espaço de estados rotulado com cada subfórmula que compõe a fórmula CTL avaliada.

Algoritmo 2: Satisfação de fórmulas CTL em ENF - Sat(Φ) adaptado de Baier e Katoen [2]

Entrada: sistema de transição finito *TS* com conjunto de estados S, fórmula Φ em ENF, lista Labels vazia
Saída: $Sat(\Phi) = \{s \in S \mid s \models \Phi\}$, lista Labels preenchida
`/* computação recursiva dos conjuntos` $Sat(\Psi)$ `para todas sub-fórmulas` Ψ `de` Φ `*/`
início
  caso Φ
    selecione $true$ **faça**
      retorna S;
    fim selec
    selecione a **faça**
      Labels $\leftarrow$ Labels $\cup$ { $(a \rightarrow \{s \mid s \in S \wedge s \models a\})$ };
      retorna $\{s \in S \mid a \in L(s)\}$;
    fim selec
    selecione $\Phi_1 \wedge \Phi_2$ **faça**
      T $:= Sat(\Phi_1) \cap Sat(\Phi_2)$
      Labels $\leftarrow$ Labels $\cup$ { ($(\Phi_1 \wedge \Phi_2) \rightarrow T$) };
      retorna T;
    fim selec
    selecione $\neg\Psi$ **faça**
      T $:= S \setminus Sat(\Psi)$
      Labels $\leftarrow$ Labels $\cup$ { ($\neg\Psi \rightarrow T$) };
      retorna T;
    fim selec
    selecione $\exists \bigcirc \Psi$ **faça**
      T $:= \{s \in S \mid Post(s) \cap Sat(\Psi) \neq \varnothing\}$
      Labels $\leftarrow$ Labels $\cup$ { ($\exists \bigcirc \Psi \rightarrow T$) };
      retorna T;
    fim selec
    selecione $\exists(\Phi_1 \cup \Phi_2)$ **faça**
      `/* Computa o menor ponto-fixo */`
      $T := Sat(\Phi_2)$;
      enquanto $\{s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing\} \neq \varnothing$ **faça**
        seja $s \in \{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \}$;
        $T := T \cup \{ s \}$;
      fim enqto
      Labels $\leftarrow$ Labels $\cup$ { ($\exists(\Phi_1 \cup \Phi_2) \rightarrow T$) };
      retorna T;
    fim selec
    selecione $\exists\Box\Psi$ **faça**
      `/* Computa o maior ponto-fixo */`
      $T := Sat(\Psi)$;
      enquanto $\{s \in T \mid Post(s) \cap T = \varnothing\} \neq \varnothing$ **faça**
        seja $s \in \{ s \in T \mid Post(s) \cap T = \varnothing \}$;
        $T := T \setminus \{ s \}$;
      fim enqto
      Labels $\leftarrow$ Labels $\cup$ { ($\exists\Box\Psi \rightarrow T$) };
      retorna T;
    fim selec
  fim selec
fim

S: Espaço de estados atingíveis (RSS)
$Post(s)$: Estados posteriores (sucessores) de um estado s
$L(s)$: Conjunto de $labels$, ou seja, proposições atômicas que são satisfeitas em um estado s
f : subfórmula que compõe a fórmula Φ
$Labels()$: associa a cada (sub)fórmula Ψ de Φ os estados $\in \{S\}$ que a satisfazem

---

O Algoritmo 3 representa o *Verificador de Estado Inicial* (ver Figura 4.1). Ele é responsável por verificar se existe um estado inicial do modelo que satisfaça a fórmula Φ avaliada e por inserir este estado na lista *trace*. Este método tem como entrada o sistema de transição de estados TS, o conjunto I de estados iniciais, a especificação Φ e a lista *Labels*. Na linha 4 é feita a interseção entre o conjunto de estados iniciais e o conjunto de estados que satisfazem a fórmula Φ, recuperados por *Labels(Φ)*, deste modo um estado inicial s é encontrado e inserido no *trace*, linha 6, dando início assim, a construção de uma testemunha para a especificação avaliada.

Algoritmo 3: iniciaTrace(TS, I, Fórmula Φ, Labels)

```
   Entrada: sistema de transição de estados TS, conjunto de estados iniciais I,
            fórmula Φ, Labels
   Saída: trace com o estado inicial que compõe a testemunha
   Dados: trace
   Dados: MDD T
 1 início
 2     trace.novo();
       /* interseção entre o conjunto de estados iniciais I e o
       conjunto de estados em Labels(Φ)                              */
 3     seja TS = (S, I, R, L)
 4         T := I ∩ Labels(Φ);
 5         seja s ∈ T
 6             trace.insere(s);
 7             retorna trace;
 8
 9
10 fim
11 ____________________________________________________________________
12 trace: lista ordenada de estados.
13     novo: cria uma nova lista vazia.
14     insere: insere um novo estado s após o último estado em trace.
15
```

A construção da testemunha é realizada através do caminhamento em pré-ordem na árvore sintática que representa a fórmula Φ avaliada, ou seja, visita a raiz, o filho da esquerda e então o filho da direita.

Conforme abordado na Seção 2.4, cada tipo de operador de caminho possui um formato de testemunha específico, dessa forma, eles são tratados pelo *Gerador de Testemunha*, Algoritmo 4, da seguinte forma:

- $\Phi_1 \wedge \Phi_2$ - para fórmulas compostas pela conjunção ($\wedge$), o gerador de testemunha verifica qual dos ramos da fórmula possui o maior número de operadores temporais e faz uma chamada recursiva com o ramo escolhido.

- $\neg\Psi$ - para este caso o algoritmo faz uma chamada recursiva para a fórmula Ψ.

- $\exists \bigcirc \Psi$ - para montar uma testemunha para a fórmula $\exists \bigcirc \Psi$ o gerador de testemunha tem de encontrar um sucessor do último estado s armazenado na lista *trace*, pois este estado armanezado na lista já satisfaz Φ (linha 2, ou seja, $\Phi = \exists \bigcirc \Psi$), que satisfaça Ψ. Para tanto, o algoritmo armaneza no conjunto T os estados sucessores de s, linha 4 do algoritmo. Logo após, na linha 5, o algoritmo faz a interseção entre o conjunto de estados T e o conjunto de estados codificados por Labels(Ψ). Dessa forma é encontrado um estado s' que satisfaz Ψ e que pertence ao conjunto T. Em seguida, o estado encontrado é inserido na lista *trace*, linha 7. Após o término destas computações o método *mountTrace* é chamado recursivamente para a fórmula Ψ.

- $\exists (\Phi_1 \cup \Phi_2)$ - para fórmulas compostas pelo operador $\cup$, o algoritmo separa inicialmente em conjuntos diferentes os estados que satisfazem: $\Phi_1 \cup \Phi_2$, Φ_1 e Φ_2. Na linha 23, o conjunto P recebe o conjunto de estados codificados por Labels($\Phi_1 \cup \Phi_2$). Na linha 24, o conjunto K recebe o valor que codifica os estados resultantes interseção entre o conjunto de estados codificados por Labels(Φ_2) e o conjunto P. Na linha 25, o conjunto R recebe o valor que codifica os estados resultantes da interseção entre o conjunto de estados codificados por Labels(Φ_1) e o estado P, dessa forma, ambos os conjuntos guardam somente aqueles estados que satisfazem Φ_2 e Φ_1 e que também estejam no conjunto de estados que satisfaz $\Phi_1 \cup \Phi_2$. Isso se faz necessário para garantir que o algoritmo irá considerar, na construção da testemunha, apenas aqueles estados que realmente possam levar à construção de um resultado.

  Logo após, na linha 26, o último estado armazenado na lista *trace* é recuperado e armazenado na variável s. A partir dai, começa a construção da testemunha para a fórmula. O algoritmo verifica se o estado recuperado da lista *trace* pertence ao conjunto de estados codificados por Labels(Φ_1), linha 27. Em caso positivo, o algoritmo guarda em T o conjunto de sucessores deste estado, linha 29.

  O algoritmo faz a interseção entre o conjunto T de estados e o conjunto de estados codificados por Labels(Φ_2) a fim de verificar se um destes estados armazenados em T está no conjunto de estados codificados por Labels(Φ_2), linhas 30 e 31. Em caso positivo, um destes estados

encontrados é recuperado e armazenado na lista *trace*, linha 33. O algoritmo encerra a execução, pois uma testemunha já foi encontrada. Em caso negativo, caso o estado não tenha sucessores no conjunto Labels(Φ_2), o algoritmo faz a interseção entre o conjunto T e o conjunto codificado por Labels(Φ_1) para recuperar um sucessor que esteja no conjunto codificado por Labels(Φ_1) e que não tenha sido inserido em *trace*, linha 36, 37 e 38, respectivamente. Isso evita que ciclos sejam percorridos na construção da testemunha para o operador $\cup$.

O algoritmo vai percorrendo os estados que estão em Labels(Φ_1) até que um estado que esteja em Labels(Φ_2) seja alcançado, dessa forma uma testemunha de $\Phi_1 \cup \Phi_2$ é encontrada.

- $\exists\Box\Psi$ - para construir uma testemunha para uma fórmula composta pelo operador $\Box$, o gerador de testemunhas precisa achar um caminho $s_0 s_1 .. s_n$ que contenha um ciclo. Para este tipo de fórmula, o algoritmo armazena no conjunto T os estados sucessores ao último estado s da lista *trace* que estejam no conjunto de estados codificados por Labels($\exists\Box\Psi$), linha 13. Logo após, um destes estados encontrados é inserido na lista *trace*, linha 15. As computações da linha 12 à 15 são repetidas até que um ciclo seja detectado, caracterizando assim a testemunha para o operador $\Box$. Após o término destas computações, o método *mountTrace* é chamado recursivamente para a fórmula Ψ.

Algoritmo 4: mountTrace(formula Φ, Labels, TS, trace)

Entrada: fórmula Φ, Labels - conjunto de rótulos das subfórmulas de Φ, Sistema de transição de estados TS, *trace* - lista com os estados que compõe a testemunha
Saída: Um *trace* de execução do modelo que representa a testemunha para fórmula avaliada
Dados: aut_st s;
Dados: MDD T, P, K, R, Q;

```
1  início
2      caso Φ
3          selecione ∃ ◯ Ψ faça
4              T : = Post(trace.consulta()) ;
5              T := T ∩ Labels(Ψ);
6              seja s um estado de T
7                  trace.insere(s);
8              mountTrace(Ψ, Labels, TS, trace);
9          fim selec
10         selecione ∃□Ψ faça
11             enquanto true faça
12                 T := Post(trace.consulta())
13                 T := T ∩ Labels(∃□Ψ);
14                 seja s um estado de T
15                     trace.insere(s);
16                 se trace.temCiclo() então
17                     para;
18                 fim se
19             fim enqto
20             mountTrace(Ψ, Labels, TS, trace);
21         fim selec
22         selecione ∃(Φ1 ∪ Φ2) faça
23             P := Labels(Φ1 ∪ Φ2);
24             K := Labels(Φ2) ∩ P;
25             R := Labels(Φ1) ∩ P;
26             s := trace.consulta();
27             se s ∈ Labels(Φ1) então
28                 enquanto true faça
29                     T := Post(s);
                       /* Verifica se há algum estado em T que pertença a K (Labels(Φ2))   */
30                     Q := T ∩ K;
31                     se Q ≠ ∅ então
32                         seja s um estado de Q
33                             trace.insere(s);
34                             vai para fim enqto;
35                     fim se
                       /* Se não há, pega um estado que satisfaça Φ1                       */
36                     T := T ∩ R;
37                     seja s ∈ T | s ∉ trace.estados()
38                         trace.insere(s);
39                 fim enqto
40             fim se
41             mountTrace(Φ2, Labels, TS, trace);
42         fim selec
43         selecione Φ1 ∧ Φ2 faça
44             se numForm(Φ1) ⩾ numForm(Φ2) então
45                 mountTrace(Φ1, Labels, TS, trace);
46             fim se
47             senão
48                 mountTrace(Φ2, Labels, TS, trace);
49             fim se
50         fim selec
51         selecione ¬Ψ faça
52             mountTrace(Ψ, Labels, TS, trace);
53         fim selec
54     fim selec
55
56     numForm(): retorna o número de operadores temporais em uma fórmula Φ
57     trace: lista ordenada de estados.
58         consulta : retorna o último estado da lista trace.
59         estados : retorna o conjunto de estados em trace.
60         bool temCiclo(): verifica se o último estado em trace se repete anteriormente.
61
62
63 fim
```

A ferramenta suporta a geração de testemunhas lineares. Considerando-se isso, durante a implementação, foram identificados *pontos de parada*, ou seja, situações em que o resultado a ser gerado deixava de ter formato de *trace* de execução e passava a ter formato de uma árvore de execução do sistema ou ainda situações em que o resultado de retorno é o espaço de estados inteiro do modelo o que é inviável para apresentação. A seguir são relacionados os *pontos de parada* identificados durante a implementação:

1. Testemunha para um operador existencial negado - uma testemunha para uma fórmula do tipo $\neg\exists\Phi$ teria de mostrar todas as execuções do modelo para assegurar que não há caminhos dentro dele que satisfaçam $\exists\Phi$, que torna inviável sua apresentação. Neste caso, somente o estado inicial que satisfaz a fórmula é exibido ao usuário.

2. Fórmulas com o operador $\cup$ - fórmulas em que o lado esquerdo do operador $\cup$ seja uma sub-fórmula de caminho, como em $\exists(\exists\Phi \cup \Psi)$, força que, para cada estado que satisfaça o lado esquerdo do operador, exista um posterior a ele que satisfaz Φ. Para esses casos, o algoritmo apenas irá mostrar os estados que satifaçam $\exists\Phi$, sem mostrar os posteriores a este, até que um estado que satisfaça Ψ seja encontrado.

3. Fórmulas aninhadas com o operador $\Box$ - fórmulas aninhadas com formato $\exists\Box\exists\Phi$, ou seja, com o operador *Globally* como operador mais externo forçam que a partir de cada estado que satisfaz a fórmula $\exists\Box$ exista um caminho que satisfaça a fórmula interna $\exists\Phi$. Este tipo de fórmula possui um formato de testemunha, com vários ramos, não suportado pela ferramenta. Desse modo, o tratamento dado a ela é o de construir o *trace* com os estados que satisfaçam $\exists\Box$, sem abrir outros caminhos que satisfaçam a fórmula $\exists\Phi$, e apenas marcando que os estados do *trace* satisfazem $\exists\Phi$.

4. Fórmulas com a conjunção *and* - fórmulas como, por exemplo, $\exists\Phi \wedge \exists\Psi$, também possuem a característica de ter como resultado uma árvore de execução. Contudo, para este tipo de fórmula uma abordagem diferente foi definida. Quando o algoritmo encontra uma fórmula com a conjunção *and*, ele verifica qual dos dois ramos do operador possui o maior número de operadores temporais e constroi uma testemunha para este. Essa abordagem visa mostrar ao usuário a testemunha mais completa possível. Caso ambos os lados possuam o mesmo número de operadores temporais, a testemunha é criada com o ramo da esquerda da conjunção.

As características apresentas pelos itens 1 e 2 ratificam o fato de que contraexemplos são gerados para fórmulas quantificadas universalmente, enquanto que testemunhas são geradas para fórmulas quantificadas existencialmente [2, 12]. Para as situações supracitadas o *trace* gerado é incompleto [14], ou seja, mostra parte da satisfação ou refutação da fórmula. O fato de ser incompleto é informado ao usuário.

A implementação do Algoritmo 4 agrega ao que foi apresentado, controles para a geração de resultados lineares, evitando assim a geração de resultados ramificados, gerados nas situações

identificadas e apresentadas pelos itens 1 ao 4. Os controles memorizam ao longo da recursão de *mountTrace* se a (sub)fórmula em tratamento:

1. foi negada;
2. é sub-fórmula ao lado esquerdo do operador $\cup$;
3. é sub-fórmula do operador $\exists\Box$.

Ao longo da condução do trabalho, os formatos de especificações que levam a resultados ramificados foram identificados dentro dos *pontos de parada* supracitados. Dessa forma, com estes controles o algoritmo cobre e trata os casos em que uma testemunha para uma especificação avaliada deixa de ter formato de resultado linear e passa a ter formato de resultado em árvore.

A Figura 4.2 mostra o diagrama de atividades para a geração de uma testemunha no Verificador de Modelos. O diagrama mostra a interação entre as rotinas do verificador de modelos até a geração do resultado. A execução é baseada em três personagens: o usuário, a interface do sistema e o sistema.

O fluxo de atividade começa quando o usuário fornece ao sistema os parâmetros necessários à execução do programa. Essas informações são: o modelo SAN que será verificado, a especificação descrita por uma fórmula em CTL que será avaliada, além do tipo de saída que o usuário deseja obter (um contraexemplo ou uma testemunha).

Caso o número de parâmetros informado seja menor do que o necessário, uma mensagem é exibida pela interface ao usuário, que por sua vez pode reenviar as informações ou cancelar a execução do programa.

Passada a etapa de verificação das informações fornecidas, chega a vez do compilador, de posse do que foi informado, ser executado. Conforme descrito no Capítulo 3, ele é responsável por gerar o descritor Markoviano, utilizado pelo Gerador RSS (Gerador do Espaço de Estados Atingível) do modelo, etapa posterior.

A execução do Analisador CTL, responsável por gerar a árvore sintática da fórmula CTL é realizada logo após o Gerador RSS. A árvore sintática gerada nesta etapa é utilizada pelo Conversor ENF, responsável por gerar a árvore sintática da fórmula CTL em ENF. Entre essas duas etapas é executada a rotina de Rotulação de Estados, responsável pela rotulação do espaço de estados com as proposições atômicas verdadeiras em cada estado atingível do modelo.

A partir daí, o Tratator ENF é executado, levando em consideração o tipo de resultado solicitado pelo usuário. Caso o usuário opte por um contraexemplo a fórmula CTL em ENF tem de ser manipulada para que o resultado gerado equivalha a um contraexemplo da fórmula CTL dada como parâmetro para o sistema. Caso o usuário opte por uma testemunha, nenhuma mudança é feita e a fórmula é enviada para a etapa posterior.

A etapa seguinte consiste na execução do algoritmo de satisfação de fórmulas, SAT. A saída desta rotina é o conjunto de estados que satisfaz a fórmula Φ e a lista Labels, preenchida com os estados que satisfazem as (sub)fórmulas de Φ.

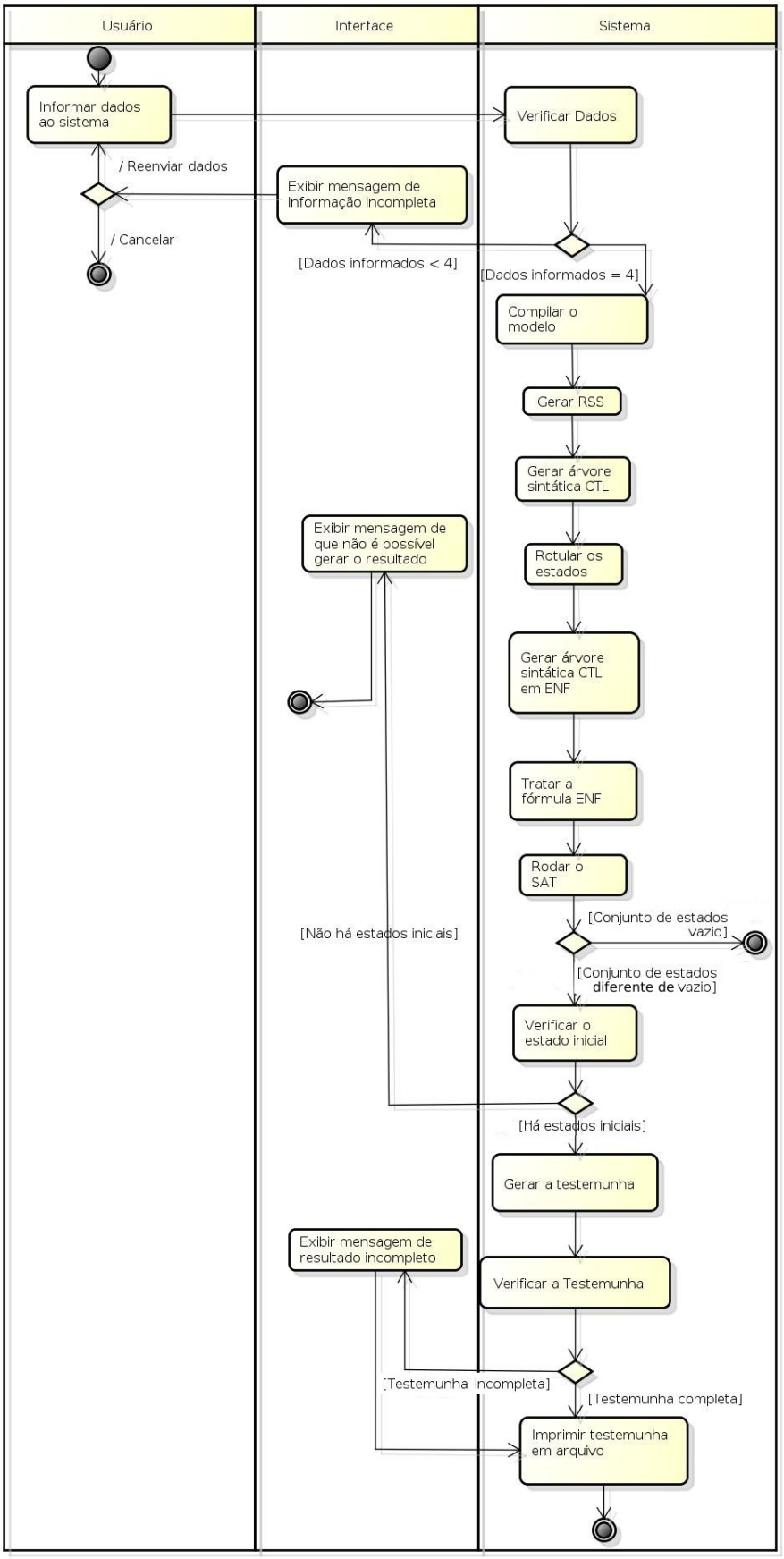


Figura 4.2 – Diagrama de atividade: Gerando uma testemunha

.

A próxima rotina a ser executada é o Verificador de Estado Inicial. A partir do conjunto de estados que satisfazem a fórmula e do conjunto de estados iniciais do modelo, esta rotina verifica se há algum estado inicial que satisfaça a fórmula. Em caso positivo, o estado encontrado será o ponto de partida na construção da testemunha, etapa seguinte.

Caso não exista um estado inicial, antes de encerrar o programa, uma mensagem é exibida ao usuário avisando que não há estados iniciais que satisfaçam a fórmula e que por isso uma testemunha não pode ser montada.

O Gerador de Testemunha constrói então, a testemunha para a fórmula avaliada, respeitando os pontos de parada e abordadens adotadas para a geração dos resultados, descritas anteriormente neste capítulo.

Note que a testemunha gerada pela ferramenta pode não ser completa. Para os casos em que ela não é completa, antes de gravar a testemunha em arquivo, o sistema mostra uma das seguintes mensagens ao usuário:

- Testemunha em formato de árvore - a mensagem *Testemunha em formato de árvore, o estado exibido faz parte da testemunha* é exibida quando o item 1 ou 2 dos *pontos de parada* ocorrer. O resultado da solicitação é todo o espaço de estados do modelo, ou seja, a árvore de execução do sistema. Nesses casos apenas o estado inicial do modelo é exibido, se este satisfizer a fórmula avaliada.
- Testemunha incompleta - a mensagem *A testemunha/contraexemplo mostra parte da satisfação/refutação da fórmula* é exibida quando:
  - Uma a negação do operador existencial aparecer numa subfórmula da fórmula CTL verificada. Nesse caso somente um estado que satisfaz a subfórmula é exibido.
  - Ocorrer a situação caracterizada pelo item 3 da lista *pontos de parada*.
  - A fórmula possuir um *and*. Nesses casos apenas um dos ramos é usado na construção da testemunha.
  - Uma fórmula com o operador *until* possuir no seu lado esquerdo uma outra fórmula de caminho.

Após exibir a mensagem o sistema imprime em arquivo a testemunha encontrada para a fórmula.

5. EXPERIMENTOS

Para analisar e demonstrar o comportamento da abordagem adotada para geração de contraexemplos e testemunhas, foram conduzidos testes com os modelos SAN apresentados ao longo desta dissertação. Modelos de diferentes tamanhos foram gerados a fim de verificar o comportamento do algoritmo. Para o modelo do Jantar dos Filósofos foram utilizados modelos com três, cinco e 14 filósofos. Para o modelo da Linha de Produção foram feitos testes com modelos de três e nove estações. O modelo de cálculo de vazão líquida de dados de uma rede wireless adhoc foi testado com modelos de dez, 13 e 15 nodos.

Ao todo, 48 propriedades em CTL foram executadas com os três modelos, destas 41 para o modelo Jantar dos Filósofos, quatro para o modelo adhoc e três para o modelo de Linha de Produção. Algumas delas têm sentido semântico, como propriedades para verificar *starvation* e *liveness* por exemplo; outras foram criadas apenas para verificar se o algoritmo apresentava o comportamento esperado dentro da estratégia e características da implementação descritas no Capítulo 4.

Com a intenção de verificar a coerência das saídas geradas pelo *SAN Model Checker*, os resultados de 18 propriedades foram comparados às saídas geradas pelo verificador de modelos NuSMV [9]. A escolha destas 18 propriedades dentre as 48 testadas, justica-se pelo fato destas conterem um número maior de operadores temporais e conectivos lógicos na sua composição, representando assim o melhor conjunto de fórmulas, dentre as definidas, para testar o algoritmo implementado. Comparar as saídas do verificador de modelos em SAN com as do NuSMV foi possível pois, dentro do projeto de construção de um verificador para modelos descritos em SAN, existe um trabalho em paralelo focado na tradução de modelos SAN para a linguagem do NuSMV *Model Checker* [34].Dessa forma, estas 18 propriedades foram avaliadas em modelos iguais em ambos os verificadores. Destas 18 propriedades, seis (propriedades 5.1, 5.4, 5.8, 5.10, 5.11 e 5.13) apresentaram o mesmo resultado, dando um forte indício da coerência da implementação da geração de contraexemplos e testemunhas para o SAN *Model Checker*. Em algumas situações, as propriedades avaliadas apresentaram resultados similares e em alguns casos, resultados diferentes, decorrentes da estratégia adotada ao longo deste trabalho.

São apresentados neste capítulo, por questões de adequação ao espaço disponível, os resultados obtidos com as propriedades testadas em modelos pequenos, isto é, para o modelo do Jantar dos Filósofos são apresentados os resultados obtidos com o modelo de três filósofos; para o modelo adhoc, são mostrados os resultados obtidos com o modelo de dez nodos e para o modelo de Linha de Produção, são apresentados resultados para o modelo de três estações. Os modelos maiores foram utilizados para verificar até que tamanho a ferramenta conseguiu verificar modelos fazendo uso do *hardware* disponível.

Para realizar os experimentos foi alocada uma máquina do LAD[1](Laboratório de Alto Desempenho da Pontifícia Universidade Católica do Rio Grande do Sul). A configuração da máquina é a que segue: *DellPowerEdge R610 com dois processadores Intel Xeon Quad-core E5520 2.27 GHz*

[1]Acesso: http://www.lad.pucrs.br

Hyper-treading e 16 GB de memória. Sistema Operacional Ubuntu (versão servidor) Linux 10.04 de 64 bits.

A seguir são apresentadas as especificações avaliadas em cada um dos modelos apresentados ao longo da dissertação. A Seção 5.2 mostra os resultados gerados pelos dois verificadores.

5.1 Propriedades CTL para SAN

Propriedades CTL para modelos SAN são expressões que definem comportamentos desejados sobre um modelo. As expressões SAN são utilizadas para definir o conjunto de proposições atômicas que compõem uma propriedade. A seguir, são listadas as propriedades definidas para cada modelo utilizado ao longo da dissertação.

5.1.1 Propriedades CTL para o Jantar dos Filósofos

Para os problema do Jantar dos Filósofos, 11 propriedades foram definidas. As proposições atômicas definidas para expressar os comportamentos desejados para as propriedades descritas são:

$$Fil_iThinking = (st\ Fil_i == Thinking);$$
$$Fil_iRight = (st\ Fil_i == Right);$$
$$Fil_iLeft = (st\ Fil_i == Left);$$
$$alguemCome = ((st\ Fil_i == Left)||(st\ Fil_{i+1} == Left)...||(st\ Fil_{N-1} == Right));$$

Onde: i representa o índice do filósofo e N o número de filósofos do modelo.

A proposição atômica $Fil_iThinking$ denota que o filósofo i está no estado *Thinking*. A proposição atômica Fil_iRight denota que o filósofo i está no estado *Right*. Já a proposição atômica Fil_iLeft denota que o filósofo i está no estado *Left* e, por fim, a proposição $alguemCome$ denota algum dos filósofos está comendo, isto é, os filósofos destros podem estar no estado *Left* ou o filósofo canhoto no estado *Right*.

Com estas proposições atômicas, foram definidas as seguintes fórmulas CTL:

$$\exists\Diamond\exists\Box Fil0 = Right \tag{5.1}$$

Esta propriedade denota que existe um futuro caminho em que o filósofo zero (Fil0) sempre fica *com fome*, ou seja, no estado *Right*. Esta propriedade denota um comportamento de *starvation* para o filósofo zero, uma vez que o estado *Left* nunca é alcançado.

$$\forall\Box(Fil0 = Left \rightarrow \forall\Diamond(Fil1 = Left \wedge Fil2 = Right)) \tag{5.2}$$

A propriedade foi criada para testar a funcionalidade do algoritmo, pois, a sua conversão para ENF resulta na fórmula: $\exists(true \cup (Fil0 = Left \wedge \exists\Box\neg(Fil1 = Left \wedge Fil2 = Right)))$ e faz com que o algoritmo, ao chegar na conjunção ($\wedge$), tenha de escolher entre um dos ramos (o que possuir o maior número de operadores temporais) para montar a testemunha. Nesse caso, o algoritmo montará um caminho com o ramo direito da conjunção.

$$\forall\Box Fil1 = Right \tag{5.3}$$

Esta propriedade verifica se em todos os caminhos de execução do modelo o filósofo 1 (Fil1) fica sempre *com fome*, ou seja, no seu estado *Right*.

$$\exists(Fil0 = Thinking \cup \exists\Box Fil1 = Right) \tag{5.4}$$

Esta propriedade também demonstra um comportamento de *starvation*, contudo ela é um subconjunto da propriedade que representaria *starvation* para o Fil1, pois, fixa que o filósofo 0 (Fil0) esteja no estado *Thinking* até que um estado que satisfaça Fil1 no estado *com fome* (*Right*) seja alcançado.

$$\neg\exists \bigcirc Fil0 = Thinking \vee \neg\exists\Diamond\exists\Box Fil1 = Right \tag{5.5}$$

Para o caso desta propriedade, foi solicitada a geração de um contraexemplo. Para tanto, as manipulações necessárias são realizadas, resultando na fórmula temporal: $\exists\bigcirc Fil0 = Thinking \wedge \exists(true \cup \exists\Box Fil1 = Right)$. Dessa forma, a execução deste teste visa mostrar o comportamento do algoritmo quando encontra um $\wedge$ na fórmula, bem como o comportamento da geração para operadores temporais aninhados, visto que, de acordo com a estratégia adotada o *trace* será montado com o ramo direito da conjunção, que possui maior número de operadores temporais.

$$\exists(true \cup \exists \bigcirc Fil0 = Thinking) \wedge \exists(true \cup \exists\Box Fil1 = Right) \tag{5.6}$$

Na propriedade acima, o algoritmo terá de escolher novamente entre um dos ramos da conjunção, contudo, nesse caso, como os dois ramos possuem o menos número de operadores temporais, o *trace* será montado com o ramo esquerdo, visto que essa foi a estratégia definida.

$$\forall\Box\forall \bigcirc Fil2 = Right \tag{5.7}$$

A propriedade define que para todo o caminho globalmente, a partir de qualquer estado todos seus sucessores alcancem, em um passo, um estado em que o Fil2 está no estado *Right*. Este caso visa mostrar o comportamento do algoritmo para busca de contraexemplos de fórmulas aninhadas compostas pelo quantificador universal.

$$\forall\Box(Fil0 = Thinking \vee Fil0 = Right \rightarrow \forall\Diamond Fil1 = Left) \tag{5.8}$$

A propriedade acima tem intenção de verificar o comportamento do algoritmo para fórmulas com operadores temporais aninhados que também possuem disjunção e implicação de proposições atômicas. Esta fórmula diz que se o filósofo 0, Fil0, está no estado *Thinking* ou *Right* isso implica em que a partir daquele estado todos os caminhos levem a um caminho futuro em que o estado *Left* do filósofo um, Fil1, é alcançado.

$$\forall\Box(Fil0 = Left \rightarrow \forall(\neg Fil0 = Left \cup Fil1 = Left)) \tag{5.9}$$

Propriedade criada para demonstrar o comportamento do algoritmo ao aninhar-se os operadores de caminho $\forall\Box$ e $\forall\cup$ com uma implicação entre eles.

$$\exists\Diamond(alguemCome) \tag{5.10}$$

A propriedade procura se existe um caminho onde um dos filósofos come, isto é, se algum dos filósofos destros alcança o estado *Left* ou o filósofo canhoto o estado *Right*.

$$\exists\Diamond\exists \bigcirc Fil0 = Left \tag{5.11}$$

Esta propriedade denota que existe um caminho futuro no qual existe um estado em que o filósofo zero (Fil0) está no estado *Left*.

5.1.2 Propriedades CTL para o modelo de rede adhoc

Para o modelo de rede adhoc quatro propriedades foram escritas para verificar o envio e recebimento de pacotes através da rede. As proposições atômicas utilizadas nas propriedades são:

$$r_i = (st\ MN_i == R)$$
$$t_i = (st\ MN_i == T)$$

Com estas proposições as seguintes propriedades foram definidas:

$$\exists\Box(MN_1 = T \rightarrow \exists\Diamond MN_N = R) \tag{5.12}$$

Esta propriedade verifica se existe um caminho no modelo adhoc onde globalmente, para todos os estados, toda transmissão resulta numa recepção futura por parte do último nodo da rede.

$$\exists(\neg MN_N = R \cup MN_1 = T) \tag{5.13}$$

Esta propriedade verifica se existe um caminho onde uma recepção por parte do último nodo do modelo não acontece até que uma transmissão pelo primeiro nodo ocorra.

$$\exists\Box\exists(true \cup MN_5 = T) \tag{5.14}$$

Propriedade criada para demonstrar o comportamento do algoritmo ao aninhar os operadores $\exists\Box$ e $\exists\cup$. Ela denota que existe um caminho que globalmente a partir de qualquer estado existe um futuro caminho em que o nodo cinco estará no estado *T*.

$$\neg\exists\Box(MN_1 = T \rightarrow \exists\Diamond MN_N = R) \tag{5.15}$$

Esta propriedade diz que não existe um caminho em que globalmente, a partir de qualquer estado, exista um outro caminho em que toda transmissão por parte do primeiro nodo, resulta numa recepção futura do último nodo.

5.1.3 Propriedades CTL para o modelo de Linha de Produção

Para o modelo de linha de produção foram definidas as proposições atômicas abaixo:

$$Estação_i Bloqueada = st\ MN_i == Bloqueada$$
$$Estação_i Desbloqueada = st\ MN_i == Desbloqueada$$

Tendo estas proposições definidas, as seguintes propriedades CTL foram descritas:

$$\exists(true \cup Estação_N Bloqueada) \tag{5.16}$$

Esta propriedade denota que há um caminho no qual a última estação está bloqueada.

$$\exists\Diamond(Estação_N Bloqueada \rightarrow \forall\Diamond Estação_N Desbloqueada) \tag{5.17}$$

Esta propriedade apresenta um comportamento de *liveness*, pois denota que, em todos os futuros caminhos, a partir de um estado de bloqueio da última estação, ela será desbloqueada. Dessa forma, a última estação não ficará infinitamente num estado de bloqueio.

$$\exists\Diamond(Estação_N Bloqueada \rightarrow \exists\Diamond Estação_N Desbloqueada) \tag{5.18}$$

Propriedade com semântica semelhante a anterior, contudo denota apenas que existe futuro caminho a partir de um estado de bloqueio da última estação em que ela será desbloqueada.

5.2 Resultados

Esta seção apresenta os resultados obtidos pelo SAN *Model Checker* e NuSMV *Model Checker* ao procurar por testemunhas ou contraexemplos para as propriedades definidas na Seção 5.1. A disposição das informações é a que segue: inicialmente é mostrada a propriedade testada e o tipo de saída solicitada (contraexemplo ou testemunha), logo depois o resultado gerado pelo SAN *Model Checker*, para então o resultado obtido com o NuSMV *Model Checker*.

Os nomes de estados e autômatos apresentados nesta seção estão de acordo com a maneira em que foram descritos os modelos SAN (consultar apêndices deste volume). Portanto, para o modelo do Jantar dos Filósofos , os resultados referenciam os filósofos como P_i, onde i representa o número do filósofo considerado. Para os resultados do modelo adhoc, os nodos pertencentes a rede são referenciados como MN_i onde i representa o número do nodo na rede, e os estados são exibidos como: I para $Idle$, T para $Transmiting$ e R para $Receiving$. Para o modelo de Linha de Produção, as estações são referenciadas como M_i, onde i é o número da estação, e o estado de bloqueio delas é representado pelo estado $1, 2$.

5.2.1 Propriedade 5.1, resultado gerado: testemunha

A propriedade 5.1 exibe um comportamento de *starvation* do filósofo zero (Fil0). Os *traces* gerados pelo SAN *Model Checker* e NuSMV *Model Checker*, Figuras 5.1 e 5.2 respectivamente, mostram que ambos os verificadores chegaram a mesma resposta.

A disposição dos filósofos no resultado gerado pelo SAN *Model Checker* é a mesma de declaração deles no modelo SAN, ou seja, Filósofo dois (Fil2), Filósofo um (Fil1) e Filósofo zero (Fil0). Ao lado de cada estado que faz parte do *trace*, o SAN *Model Checker* exibe a subfórmula satisfeita por aquele estado e o evento disparado que causa a transição para o estado da linha posterior.

Note que, para a geração no NuSMV *Model Checker*, a especificação teve de ser negada $(\neg(EF(EG\ a\_P0.state = s\_Right)))$, pois o verificador gera somente contraexemplos. Dessa forma, para buscar uma testemunha de uma fórmula quantificada existencialmente, a mesma tem de ser negada para que o NuSMV *Model Checker* gere um contraexemplo da fórmula, o que semanticamente equivale à testemunha da fórmula existencial sem a negação.

A saída gerada pelo NuSMV também foi alterada no que diz respeito aos estados locais exibidos como resultado. O verificador tem como padrão exibir somente o(s) autômato(s) que muda(m) de estado durante a transição. Na saída editada também são exibidos os autômatos cujos estados locais não mudaram durante a transição.

A testemunha gerada pelos verificadores começa no estado global *Thinking* - *Thinking* - *Thinking* e vai até o estado *Thinking* - *Thinking* - *Right*, linha 11, onde um ciclo é detectado, pois o estado já tinha sido visitado na linha oito, satisfazendo dessa maneira a fórmula interna $\exists\Box Fil0 = Right$.

```
 1 Testemunha gerada pela ferramentapara especificação: EF(EG((st P0 == Right)))
 2
 3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
 4
 5  Estado Global -  (sub)formula valida no estado -  evento disparado
 6
 7 Thinking - Thinking - Thinking -  E(True U EG((st P0 == Right))) - t_r_0
 8 Thinking - Thinking - Right -  EG((st P0 == Right)) - t_l_2
 9 Left - Thinking - Right -  EG((st P0 == Right)) - l_r_2
10 Right - Thinking - Right -  EG((st P0 == Right)) - r_t_2
11 Thinking - Thinking - Right -  EG((st P0 == Right)) -  inicio do ciclo
```

Figura 5.1 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.1

```
-- specification !(EF (EG a_P0.state = s_Right))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-- Loop starts here
-> State: 1.2 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Right
-> State: 1.3 <-
  a_P2.state = s_Left
  a_P1.state = s_Thinking
  a_P0.state = s_Right
-> State: 1.4 <-
  a_P2.state = s_Right
  a_P1.state = s_Thinking
  a_P0.state = s_Right
-> State: 1.5 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Right
```

Figura 5.2 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.1

5.2.2 Propriedade 5.2, resultado gerado: contraexemplo

Esta propriedade testa o funcionamento do algoritmo, pois, a sua conversão para ENF resulta na fórmula: $\exists(true \cup (Fil0 = Left \wedge \exists\Box\neg(Fil1 = Left \wedge Fil2 = Right)))$ e faz com que o algoritmo, ao chegar na conjunção $\wedge$, tenha de escolher entre o ramo que possui maior número de operadores temporais.

O resultado obtido pelo SAN *Model Checker* (Figura 5.3) apresenta, como contraexemplo, uma sequência de dez estados, começando pelo estado inicial *Thinking-Thinking-Thinking* e percorrendo estados que satisfazem $true$ até que um estado que satisfaz $(Fil0 = Left \wedge \exists\Box\neg(Fil1 = Left \wedge Fil2 = Right))$ é alcançado. A partir daí, o algoritmo escolhe para qual dos dois ramos irá construir o resultado. Para este caso, o ramo da direita $\exists\Box\neg(Fil1 = Left \wedge Fil2 = Right))$ é o escolhido. O algoritmo então encontra um ciclo, pois o estado $Thinking - Thinking - Thinking$ se repete nas linhas 7 e 13, respectivamente, dentro do conjunto de estados que satifaz a subfórmula $\exists\Box\neg(Fil1 = Left \wedge Fil2 = Right))$, caracterizando o resultado esperado para fórmulas com o operador $\Box$.

O resultado obtido pelo NuSMV (Figura 5.4), mostra um contraexemplo com seis estados, que inicia no mesmo estado inicial *Thinking-Thinking-Thinking* e termina com um ciclo detectado no estado *Thinking-Thinking-Left*.

```
1 Contra-exemplo gerado pela ferramenta para especificação: AG((st P0 == Left) -> AF((st P1 == Left)^(st P2 == Right)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E(True U (st P0 == Left)^EG(~((st P1 == Left)^(st P2 == Right)))) - t_l_2
8 Left - Thinking - Thinking -  True - l_r_2
9 Right - Thinking - Thinking -  True - t_r_0
10 Right - Thinking - Right -  True - r_t_2
11 Thinking - Thinking - Right -  True - r_l_0
12 Thinking - Thinking - Left -  (st P0 == Left)^EG(~((st P1 == Left)^(st P2 == Right))) - l_t_0
13 Thinking - Thinking - Thinking -  EG(~((st P1 == Left)^(st P2 == Right))) -  inicio do ciclo
```

Figura 5.3 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.2

```
-- specification AG (a_P0.state = s_Left -> AF (a_P1.state = s_Left & a_P2.state = s_Right))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.2 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Right
-- Loop starts here
-> State: 1.3 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Left
-> State: 1.4 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.5 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Right
-> State: 1.6 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Left
```

Figura 5.4 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.2

5.2.3 Propriedade 5.3, resultado gerado: contraexemplo

Esta propriedade verifica se em todos os caminhos de execução do modelo o filósofo um (Fil1) fica sempre *com fome*, ou seja, no seu estado *Right*.

O contraexemplo para esta propriedade tem de mostrar um caminho onde Fil1 vai para um estado diferente de *Right*. O resultado obtido pelo SAN *Model Checker*, Figura 5.5 mostra um contraexemplo com dois estados, aonde um estado onde o estado de Fil1 é diferente de *Right*, ou seja, não é verdade que para todos os caminhos Fil1 fica no estado *Right*.

O resultado obtido pelo NuSMV, Figura 5.6 mostra apenas um estado inicial *Thinking-Thinking-Thinking* como contraexemplo para a propriedade. Essa diferença de resultado nos contra-exemplos gerados pelos verifcadores justifica-se por diferenças na implementação de contraexemplos e testemunhas para fórmulas CTL no SAN *Model Checker*.

Note que, durante a geração de um contraexemplo, se um dos estados iniciais não satisfaz a fórmula avaliada, este por si só, já representa um contraexemplo. Esta é a razão pela qual o NuSMV *Model Checker* exibe um contraexemplo de um estado apenas. Contudo, é importante ressaltar que, o resultado gerado pelo SAN *Model Checker*, com um estado além daquele que já viola a especificação avaliada, está correto. Considerando que o algoritmo implementado restringe-se à buscas por testemunhas (uma vez que ele computa fórmulas CTL em ENF) e que por essa razão somente conjuntos de estados que satisfazem a fórmula são considerados durante a construção do resultado, não ocorrerá em resultados similares a este, uma situação em que o algoritmo exiba um estado posterior ao estado inicial, que viole o próprio contraexemplo. Dessa forma, o estado a mais, exibido pelo SAN *Model Checker*, não acarreta na violação do próprio resultado.

```
1 Contra-exemplo gerado pela ferramenta para especificação: AG((st P1 == Right))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E(True U ~((st P1 == Right))) - t_l_2
8 Left - Thinking - Thinking -  ~((st P1 == Right)) -  fim do caminho
```

Figura 5.5 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.3

```
-- specification AG a_P1.state = s_Right  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
```

Figura 5.6 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.3

5.2.4 Propriedade 5.4, resultado gerado: testemunha

Esta propriedade também apresenta um comportamento de *starvation*, contudo ela é considerada um subconjunto da propriedade que representaria *starvation* do Fil1, pois esta propriedade fixa que o Fil0 esteja no estado *Thinking* até que um ciclo onde Fil1 esteja em *Right* seja detectado.

Para esta propriedade ambos os verificadores chegaram à mesma resposta. A testemunha para esta propriedade possui cinco estados, iniciando por *Thinking-Thinking-Thinking* até que um ciclo no estado *Thinking-Right-Thinking* é detectado (estado se repete nas linhas oito e 11).

```
1 Testemunha gerada pela ferramentapara especificação: E((st P0 == Thinking) U EG((st P1 == Right)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E((st P0 == Thinking) U EG((st P1 == Right))) - t_r_1
8 Thinking - Right - Thinking -  EG((st P1 == Right)) - t_r_0
9 Thinking - Right - Right -  EG((st P1 == Right)) - r_l_0
10 Thinking - Right - Left -  EG((st P1 == Right)) - l_t_0
11 Thinking - Right - Thinking -  EG((st P1 == Right)) -  inicio do ciclo
```

Figura 5.7 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.4

```
-- specification !E [ a_P0.state = s_Thinking U (EG a_P1.state = s_Right) ]  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-- Loop starts here
-> State: 1.2 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Right
  a_P0.state = s_Thinking
-> State: 1.3 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Right
  a_P0.state = s_Right
-> State: 1.4 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Right
  a_P0.state = s_Left
-> State: 1.5 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Right
  a_P0.state = s_Thinking
```

Figura 5.8 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.4

5.2.5 Propriedade 5.5, resultado gerado: contraexemplo

Para montar um contraexemplo para a propriedade 5.5 o algoritmo implementado no SAN *Model Checker* teve de fazer a escolha entre os dois ramos que podem ser usados para a construção do resultado. A Figura 5.9 mostra que a ferramenta optou por construir o resultado utilizando o ramo da direita, isto é, $\exists(true \cup \exists\Box Fil1 = Right)$.

O resultado obtido com o NuSMV, Figura 5.10, mostra um contraexemplo menor, composto por dois estados, o estado inicial e um segundo estado onde o filósofo zero está no estado *Thinking*.

```
1 Contra-exemplo gerado pela ferramenta para especificação: ~(EX((st P0 == Thinking))) v ~(EF(EG((st P1 == Right))))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  EX((st P0 == Thinking))^E(True U EG((st P1 == Right))) - t_r_1
8 Thinking - Right - Thinking -  EG((st P1 == Right)) - t_r_0
9 Thinking - Right - Right -  EG((st P1 == Right)) - r_l_0
10 Thinking - Right - Left -  EG((st P1 == Right)) - l_t_0
11 Thinking - Right - Thinking -  EG((st P1 == Right)) -  inicio do ciclo
```

Figura 5.9 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.5

```
-- specification (!(EX a_P0.state = s_Thinking) | !(EF (EG a_P1.state = s_Right)))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.2 <-
  a_P2.state = s_Left
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
```

Figura 5.10 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.5

5.2.6 Propriedade 5.6, resultado gerado: testemunha

Para montar a testemunha para a propriedade 5.6 o SAN *Model Checker* novamente teve de escolher entre os ramos que podem ser usados para gerar o resultado. Neste caso em específico, como os dois ramos possuem o mesmo número de operadores temporais o algoritmo irá optar por montar a testemunha utilizando o ramo esquerdo da conjunção.

O resultado gerado pelo NuSMV, Figura 5.12, mostra que o verificador optou por gerar a testemunha utilizando o ramo direito da conjunção.

```
1 Testemunha gerada pela ferramentapara especificação: E(true U EX((st P0 == Thinking)))^E(true U EG((st P1 == Right)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E(True U EX((st P0 == Thinking)))^E(True U EG((st P1 == Right))) - t_l_2
8 Left - Thinking - Thinking -  EX((st P0 == Thinking)) - l_r_2
9 Right - Thinking - Thinking -  (st P0 == Thinking) -  fim do caminho
```

Figura 5.11 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.6

```
-- specification !(E [ TRUE U (EX a_P0.state = s_Thinking) ]  & E [ TRUE U (EG a_P1.state = s_Right) ] )  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-- Loop starts here
-> State: 1.2 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Right
  a_P0.state = s_Thinking
-> State: 1.3 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Right
  a_P0.state = s_Right
-> State: 1.4 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Right
  a_P0.state = s_Left
-> State: 1.5 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Right
  a_P0.state = s_Thinking
```

Figura 5.12 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.6

5.2.7 Propriedade 5.7, resultado gerado: contraexemplo

Os contraexemplos obtidos para a propriedade 5.7 por ambos os verificadores foi semelhantes. Tanto o SAN *Model Checker* quanto o NuSMV mostraram caminhos que alcançaram um estado em que o filósofo dois, Fil2, tem seu estado local diferente de *Right* (Figuras 5.13 e 5.14, respectivamente).

```
1 Contra-exemplo gerado pela ferramenta para especificação: AG(AX((st P2 == Right)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E(True U EX(~((st P2 == Right)))) - t_l_2
8 Left - Thinking - Thinking -  EX(~((st P2 == Right))) - t_r_0
9 Left - Thinking - Right -  ~((st P2 == Right)) -  fim do caminho
```

Figura 5.13 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.7

```
-- specification AG (AX a_P2.state = s_Right)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.2 <-
  a_P2.state = s_Left
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
```

Figura 5.14 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.7

5.2.8 Propriedade 5.8, resultado gerado: contraexemplo

Os contraexemplos gerados pelos verificadores para a propriedade 5.8 foram semelhantes para esta propriedade. A diferença está no fato de que o resultado gerado pelo SAN *Model Checker* tem um estado a mais e de que o ciclo foi detectado em estados diferentes pelos verificadores. Enquanto o SAN *Model Checker* encontrou um ciclo em $Thinking - Thinking - Thinking$, estado que se repete nas linhas sete e dez, o NuSMV chegou à mesma resposta, detectando um ciclo no estado $Thinking - Thinking - Thinking$ estados 1.1 e 1.4 da Figura 5.16, respectivamente.

```
1 Contra-exemplo gerado pela ferramenta para especificação: AG((st P0 == Thinking) v (st P0 == Right) -> AF((st P1 == Left)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E(True U ~(~((st P0 == Thinking))^~((st P0 == Right)))^EG(~((st P1 == Left)))) - t_l_2
8 Left - Thinking - Thinking -  ~(~((st P0 == Thinking))^~((st P0 == Right)))^EG(~((st P1 == Left))) - l_r_2
9 Right - Thinking - Thinking -  EG(~((st P1 == Left))) - r_t_2
10 Thinking - Thinking - Thinking -  EG(~((st P1 == Left))) -  inicio do ciclo
```

Figura 5.15 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.8

```
-- specification AG ((a_P0.state = s_Thinking | a_P0.state = s_Right) -> AF a_P1.state = s_Left)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.2 <-
  a_P2.state = s_Left
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.3 <-
  a_P2.state = s_Right
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.4 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
```

Figura 5.16 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.8

5.2.9 Propriedade 5.9, resultado gerado: contraexemplo

Esta propriedade demonstra o comportamento do algoritmo ao tratar fórmulas com operadores temporais aninhados onde existe uma implicação entre eles. A conversão desta fórmula para ENF resulta em: $\exists(true \cup (Fil0 = Left \wedge \neg(\neg(\exists\neg Fil1 = Left \cup Fil0 = Left \wedge \neg Fil1 = Left)) \wedge \neg\exists\Box Fil1 = Left))$.

O resultado gerado pelo SAN *Model Checker* mostra uma sequência de estados que começa pelo estado inicial $Thinking - Thinking - Thinking$ e termina no estado $Thinking - Thinking - Left$ que satisfaz a propriedade ao lado direito do operador $\cup$. O algoritmo não continua percorrendo a árvore que representa a propriedade, pois, ele encontra na sequência propriedades negadas, que geram um resultado em árvore, não suportado pela ferramenta.

O NuSMV mostra um contraexemplo menor, de três passos, começando pelo estado $Thinking - Thinking - Thinking$ até que um estado $Thinking - Thinking - Left$ é alcançado.

```
1 Contra-exemplo gerado pela ferramenta para especificação: AG((st P0 == Left) -> A(~((st P0 == Left)) U (st P1 == Left)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E(True U (st P0 == Left)^~(~(E(~((st P1 == Left)) U (st P0 == Left)^~((st P1 == Left))))^~(EG(~
  ((st P1 == Left)))))) - t_l_2
8 Left - Thinking - Thinking -  True - l_r_2
9 Right - Thinking - Thinking -  True - t_r_0
10 Right - Thinking - Right -  True - r_t_2
11 Thinking - Thinking - Right -  True - r_l_0
12 Thinking - Thinking - Left -  (st P0 == Left)^~(~(E(~((st P1 == Left)) U (st P0 == Left)^~((st P1 == Left))))^~(EG(~((st P1 ==
  Left))))) -  fim do caminho
```

Figura 5.17 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.9

```
-- specification AG (a_P0.state = s_Left -> A [ a_P0.state != s_Left U a_P1.state = s_Left ] )  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.2 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Right
-> State: 1.3 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Left
```

Figura 5.18 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.9

5.2.10 Propriedade 5.10, resultado gerado: testemunha

A propriedade procura se existe um caminho onde um dos filósofos come, isto é, se algum dos filósofos destros alcança o estado *Left* ou o filósofo canhoto o estado *Right*.

As Figuras 5.19 e 5.20 mostram os resultados obtidos pelo SAN *Model Checker* e NuSMV respectivamente. Para esta propriedade ambos os verificadores proveram o mesmo resultado que mostra um caminho onde o filósofo dois, Fil2, alcança o estado *Right*.

```
1 Testemunha gerada pela ferramentapara especificação: EF( ((st P0 == Left) || (st P1 == Left) || (st P2 == Right)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E(True U  ((st P0 == Left) || (st P1 == Left) || (st P2 == Right))) - t_l_2
8 Left - Thinking - Thinking -  True - l_r_2
9 Right - Thinking - Thinking -   ((st P0 == Left) || (st P1 == Left) || (st P2 == Right)) -  fim do caminho
```

Figura 5.19 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.10

```
-- specification !(EF ((a_P0.state = s_Left | a_P1.state = s_Left) | a_P2.state = s_Right))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.2 <-
  a_P2.state = s_Left
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
-> State: 1.3 <-
  a_P2.state = s_Right
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking
```

Figura 5.20 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.10

5.2.11 Propriedade 5.11, resultado gerado: testemunha

A propriedade 5.11 define que existe um caminho futuro onde é alcançado um estado que possui como sucessor um estado onde o Filósofo zero, Fil0, está no seu estado *Left*.

Para esta propriedade o resultado encontrado pelo SAN *Model Checker* e pelo NuSMV, Figuras 5.21 e 5.22, foi o mesmo. Uma testemunha com três estados, iniciando no estado inicial *Thinking-Thinking-Thinking* até um estado onde o Filósofo zero alcança o estado *Left* ser alcançado.

```
1 Testemunha gerada pela ferramentapara especificação: EF(EX((st P0 == Left)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 Thinking - Thinking - Thinking -  E(True U EX((st P0 == Left))) - t_r_0
8 Thinking - Thinking - Right -  EX((st P0 == Left)) - r_l_0
9 Thinking - Thinking - Left -  (st P0 == Left) -  fim do caminho
```

Figura 5.21 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.11

```
-- specification !(EF (EX a_P0.state = s_Left))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Thinking

-> State: 1.2 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Right

-> State: 1.3 <-
  a_P2.state = s_Thinking
  a_P1.state = s_Thinking
  a_P0.state = s_Left
```

Figura 5.22 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.11

5.2.12 Propriedade 5.12, resultado gerado: testemunha

A propriedade 5.12 verifica se existe um futuro caminho onde globalmente, a partir de cada estado que compõe o caminho, exista um outro caminho onde uma transmissão de um pacote por parte do primeiro nodo da rede, implica que exista um caminho futuro onde o último nodo recebe o que foi transmitido.

Como a propriedade possui uma fórmula mais externa composta pelo operador □ o algoritmo cria uma testemunha usando os estados que compõem a fórmula com o operador □ sem criar *traces* para a fórmula interna.

O resultado gerado pelo SAN *Model Checker* (Figura 5.23), mostra uma testemunha com 33 estados onde um ciclo é encontrado nos estados que estão nas linhas 21 e 39 respectivamente.

O resultado gerado pelo NuSMV (Figura 5.24), mostra uma testemunha com 19 estados, onde um ciclo foi detectado no estado onde todos os nodos estão em I, estados 1.1 e 1.19 respectivamente.

```
 1 Testemunha gerada pela ferramentapara especificação: EG( (st MN_1 == T) -> EF( (st MN_10 == R)))
 2
 3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
 4
 5  Estado Global -  (sub)formula valida no estado -  evento disparado
 6
 7 I - I - I - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g12
 8 T - R - I - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx1
 9 I - R - I - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g23
10 I - T - R - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx2
11 I - I - R - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g34
12 I - I - T - R - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx3
13 I - I - I - R - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g45
14 I - I - I - T - R - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx4
15 I - I - I - I - R - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g12
16 T - R - I - I - R - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx1
17 I - R - I - I - R - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g56
18 I - R - I - I - T - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx5
19 I - R - I - I - I - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g23
20 I - T - R - I - I - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx2
21 I - I - R - I - I - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g67
22 I - I - R - I - I - T - R - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx6
23 I - I - R - I - I - I - R - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g34
24 I - I - T - R - I - I - R - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx3
25 I - I - I - R - I - I - R - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g78
26 I - I - I - R - I - I - T - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx7
27 I - I - I - R - I - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g45
28 I - I - I - T - R - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx4
29 I - I - I - I - R - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g12
30 T - R - I - I - R - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx1
31 I - R - I - I - R - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g89
32 I - R - I - I - R - I - I - T - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx8
33 I - R - I - I - R - I - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g56
34 I - R - I - I - T - R - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx5
35 I - R - I - I - I - R - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g23
36 I - T - R - I - I - R - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx2
37 I - I - R - I - I - R - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g910
38 I - I - R - I - I - R - I - I - T - R -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx9
39 I - I - R - I - I - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) -  inicio do ciclo
```

Figura 5.23 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.12

```
-- specification !(EG (a_MN_1.state = s_T -> EF a_MN_10.state = s_R))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.2 <-
  a_MN_1.state = s_T
  a_MN_2.state = s_R
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.3 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_R
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.4 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_T
  a_MN_3.state = s_R
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.5 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_R
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.6 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_T
  a_MN_4.state = s_R
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.7 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_R
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.8 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_T
  a_MN_5.state = s_R
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.9 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_R
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.10 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_T
  a_MN_6.state = s_R
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.11 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_R
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.12 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_T
  a_MN_7.state = s_R
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.13 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_R
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.14 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_T
  a_MN_8.state = s_R
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.15 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_R
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.16 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_T
  a_MN_9.state = s_R
  a_MN_10.state = s_I
-> State: 1.17 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_R
  a_MN_10.state = s_I
-> State: 1.18 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_T
  a_MN_10.state = s_R
```

Figura 5.24 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.12

5.2.13 Propriedade 5.13, resultado gerado: testemunha

O resultado gerado por ambos os verificadores, Figura 5.25 e 5.26 foi o mesmo, uma testemunha contendo dois estados. A propriedade verifica se existe um caminho onde uma recepção por parte do último nodo não acontece até que um envio tenha sido feito pelo primeiro nodo.

```
1 Testemunha gerada pela ferramentapara especificação: E(~( (st MN_10 == R)) U  (st MN_1 == T))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 I - I - I - I - I - I - I - I - I - I -  E(~( (st MN_10 == R)) U  (st MN_1 == T)) - g12
8 T - R - I - I - I - I - I - I - I - I -   (st MN_1 == T) -  fim do caminho
```

Figura 5.25 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.13

```
-- specification !E [ !(a_MN_10.state = s_R) U a_MN_1.state = s_T ]   is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_MN_1.state = s_I
  a_MN_2.state = s_I
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
-> State: 1.2 <-
  a_MN_1.state = s_T
  a_MN_2.state = s_R
  a_MN_3.state = s_I
  a_MN_4.state = s_I
  a_MN_5.state = s_I
  a_MN_6.state = s_I
  a_MN_7.state = s_I
  a_MN_8.state = s_I
  a_MN_9.state = s_I
  a_MN_10.state = s_I
```

Figura 5.26 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.13

5.2.14 Propriedade 5.14, resultado gerado: testemunha

A propriedade 5.14 verifica se existe um caminho em que no futuro globalmente, a partir de cada estado, exista um caminho que levará a um estado em que o nodo MN_5 transmite um pacote. De acordo com a estratégia adotada para fórmulas que possuem o operador $\Box$ como operador mais externo, o *Gerador de Testemunhas* irá gerar um resultado com os estados que satisfazem a fórmula composta pelo operador $\Box$, sem abrir para cada um destes estados caminhos que satisfaçam a fórmula interna $\exists(true \cup MN_5 = T)$.

O resultado gerado pelo SAN *Model Checker* Figura 5.27, mostra uma testemunha com 33 estados, com um ciclo sendo detectado em $I-I-R-I-I-R-I-I-I-I$, estado que se repete nas linhas 21 e 39.

Já o resultado gerado pelo NuSMV Figura 5.28, mostra uma testemunha com 19 estados e um ciclo detectado nos estados 1.1 e 1.19.

```
1 Testemunha gerada pela ferramentapara especificação: EG(E(true U  (st MN_5 == T)))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 I - I - I - I - I - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - g12
8 T - R - I - I - I - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - tx1
9 I - R - I - I - I - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - g23
10 I - T - R - I - I - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - tx2
11 I - I - R - I - I - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - g34
12 I - I - T - R - I - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - tx3
13 I - I - I - R - I - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - g45
14 I - I - I - T - R - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - tx4
15 I - I - I - I - R - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - g12
16 T - R - I - I - I - R - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - tx1
17 I - R - I - I - R - I - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - g56
18 I - R - I - I - I - T - R - I - I - I -  EG(E(True U  (st MN_5 == T))) - tx5
19 I - R - I - I - I - I - R - I - I - I -  EG(E(True U  (st MN_5 == T))) - g23
20 I - T - R - I - I - I - R - I - I - I -  EG(E(True U  (st MN_5 == T))) - tx2
21 I - I - R - I - I - R - I - I - I - I -  EG(E(True U  (st MN_5 == T))) - g67
22 I - I - R - I - I - I - T - R - I - I -  EG(E(True U  (st MN_5 == T))) - tx6
23 I - I - R - I - I - I - I - R - I - I -  EG(E(True U  (st MN_5 == T))) - g34
24 I - I - T - R - I - I - R - I - I - I -  EG(E(True U  (st MN_5 == T))) - tx3
25 I - I - I - R - I - I - R - I - I - I -  EG(E(True U  (st MN_5 == T))) - g78
26 I - I - I - R - I - I - I - T - R - I -  EG(E(True U  (st MN_5 == T))) - tx7
27 I - I - I - R - I - I - I - R - I - I -  EG(E(True U  (st MN_5 == T))) - g45
28 I - I - I - T - R - I - I - R - I - I -  EG(E(True U  (st MN_5 == T))) - tx4
29 I - I - I - I - R - I - I - R - I - I -  EG(E(True U  (st MN_5 == T))) - g12
30 T - R - I - I - R - I - I - R - I - I -  EG(E(True U  (st MN_5 == T))) - tx1
31 I - R - I - I - R - I - I - R - I - I -  EG(E(True U  (st MN_5 == T))) - g89
32 I - R - I - I - R - I - I - T - R - I -  EG(E(True U  (st MN_5 == T))) - tx8
33 I - R - I - I - R - I - I - I - R - I -  EG(E(True U  (st MN_5 == T))) - g56
34 I - R - I - I - T - R - I - I - R - I -  EG(E(True U  (st MN_5 == T))) - tx5
35 I - R - I - I - I - R - I - I - R - I -  EG(E(True U  (st MN_5 == T))) - g23
36 I - T - R - I - I - R - I - I - R - I -  EG(E(True U  (st MN_5 == T))) - tx2
37 I - I - R - I - I - R - I - I - R - I -  EG(E(True U  (st MN_5 == T))) - g910
38 I - I - R - I - I - R - I - I - T - R -  EG(E(True U  (st MN_5 == T))) - tx9
39 I - I - R - I - I - R - I - I - I - I -  EG(E(True U  (st MN_5 == T))) -  inicio do ciclo
```

Figura 5.27 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.14

```
-- specification !(EG E [ TRUE U a_MN_5.state = s_T ] )  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-        -> State: 1.7 <-        -> State: 1.13 <-      -> State: 1.19 <-
  a_MN_1.state = s_I      a_MN_1.state = s_I      a_MN_1.state = s_I     a_MN_1.state = s_I
  a_MN_2.state = s_I      a_MN_2.state = s_I      a_MN_2.state = s_I     a_MN_2.state = s_I
  a_MN_3.state = s_I      a_MN_3.state = s_I      a_MN_3.state = s_I     a_MN_3.state = s_I
  a_MN_4.state = s_I      a_MN_4.state = s_R      a_MN_4.state = s_I     a_MN_4.state = s_I
  a_MN_5.state = s_I      a_MN_5.state = s_I      a_MN_5.state = s_I     a_MN_5.state = s_I
  a_MN_6.state = s_I      a_MN_6.state = s_I      a_MN_6.state = s_I     a_MN_6.state = s_I
  a_MN_7.state = s_I      a_MN_7.state = s_I      a_MN_7.state = s_R     a_MN_7.state = s_I
  a_MN_8.state = s_I      a_MN_8.state = s_I      a_MN_8.state = s_I     a_MN_8.state = s_I
  a_MN_9.state = s_I      a_MN_9.state = s_I      a_MN_9.state = s_I     a_MN_9.state = s_I
  a_MN_10.state = s_I     a_MN_10.state = s_I     a_MN_10.state = s_I    a_MN_10.state = s_I
-> State: 1.2 <-        -> State: 1.8 <-        -> State: 1.14 <-
  a_MN_1.state = s_T      a_MN_1.state = s_I      a_MN_1.state = s_I
  a_MN_2.state = s_R      a_MN_2.state = s_I      a_MN_2.state = s_I
  a_MN_3.state = s_I      a_MN_3.state = s_I      a_MN_3.state = s_I
  a_MN_4.state = s_I      a_MN_4.state = s_T      a_MN_4.state = s_I
  a_MN_5.state = s_I      a_MN_5.state = s_R      a_MN_5.state = s_I
  a_MN_6.state = s_I      a_MN_6.state = s_I      a_MN_6.state = s_I
  a_MN_7.state = s_I      a_MN_7.state = s_I      a_MN_7.state = s_T
  a_MN_8.state = s_I      a_MN_8.state = s_I      a_MN_8.state = s_R
  a_MN_9.state = s_I      a_MN_9.state = s_I      a_MN_9.state = s_I
  a_MN_10.state = s_I     a_MN_10.state = s_I     a_MN_10.state = s_I
-> State: 1.3 <-        -> State: 1.9 <-        -> State: 1.15 <-
  a_MN_1.state = s_I      a_MN_1.state = s_I      a_MN_1.state = s_I
  a_MN_2.state = s_R      a_MN_2.state = s_I      a_MN_2.state = s_I
  a_MN_3.state = s_I      a_MN_3.state = s_I      a_MN_3.state = s_I
  a_MN_4.state = s_I      a_MN_4.state = s_I      a_MN_4.state = s_I
  a_MN_5.state = s_I      a_MN_5.state = s_R      a_MN_5.state = s_I
  a_MN_6.state = s_I      a_MN_6.state = s_I      a_MN_6.state = s_I
  a_MN_7.state = s_I      a_MN_7.state = s_I      a_MN_7.state = s_I
  a_MN_8.state = s_I      a_MN_8.state = s_I      a_MN_8.state = s_R
  a_MN_9.state = s_I      a_MN_9.state = s_I      a_MN_9.state = s_I
  a_MN_10.state = s_I     a_MN_10.state = s_I     a_MN_10.state = s_I
-> State: 1.4 <-        -> State: 1.10 <-       -> State: 1.16 <-
  a_MN_1.state = s_I      a_MN_1.state = s_I      a_MN_1.state = s_I
  a_MN_2.state = s_T      a_MN_2.state = s_I      a_MN_2.state = s_I
  a_MN_3.state = s_R      a_MN_3.state = s_I      a_MN_3.state = s_I
  a_MN_4.state = s_I      a_MN_4.state = s_I      a_MN_4.state = s_I
  a_MN_5.state = s_I      a_MN_5.state = s_T      a_MN_5.state = s_I
  a_MN_6.state = s_I      a_MN_6.state = s_R      a_MN_6.state = s_I
  a_MN_7.state = s_I      a_MN_7.state = s_I      a_MN_7.state = s_I
  a_MN_8.state = s_I      a_MN_8.state = s_I      a_MN_8.state = s_T
  a_MN_9.state = s_I      a_MN_9.state = s_I      a_MN_9.state = s_R
  a_MN_10.state = s_I     a_MN_10.state = s_I     a_MN_10.state = s_I
-> State: 1.5 <-        -> State: 1.11 <-       -> State: 1.17 <-
  a_MN_1.state = s_I      a_MN_1.state = s_I      a_MN_1.state = s_I
  a_MN_2.state = s_I      a_MN_2.state = s_I      a_MN_2.state = s_I
  a_MN_3.state = s_R      a_MN_3.state = s_I      a_MN_3.state = s_I
  a_MN_4.state = s_I      a_MN_4.state = s_I      a_MN_4.state = s_I
  a_MN_5.state = s_I      a_MN_5.state = s_I      a_MN_5.state = s_I
  a_MN_6.state = s_I      a_MN_6.state = s_R      a_MN_6.state = s_I
  a_MN_7.state = s_I      a_MN_7.state = s_I      a_MN_7.state = s_I
  a_MN_8.state = s_I      a_MN_8.state = s_I      a_MN_8.state = s_I
  a_MN_9.state = s_I      a_MN_9.state = s_I      a_MN_9.state = s_R
  a_MN_10.state = s_I     a_MN_10.state = s_I     a_MN_10.state = s_I
-> State: 1.6 <-        -> State: 1.12 <-       -> State: 1.18 <-
  a_MN_1.state = s_I      a_MN_1.state = s_I      a_MN_1.state = s_I
  a_MN_2.state = s_I      a_MN_2.state = s_I      a_MN_2.state = s_I
  a_MN_3.state = s_T      a_MN_3.state = s_I      a_MN_3.state = s_I
  a_MN_4.state = s_R      a_MN_4.state = s_I      a_MN_4.state = s_I
  a_MN_5.state = s_I      a_MN_5.state = s_I      a_MN_5.state = s_I
  a_MN_6.state = s_I      a_MN_6.state = s_T      a_MN_6.state = s_I
  a_MN_7.state = s_I      a_MN_7.state = s_R      a_MN_7.state = s_I
  a_MN_8.state = s_I      a_MN_8.state = s_I      a_MN_8.state = s_I
  a_MN_9.state = s_I      a_MN_9.state = s_I      a_MN_9.state = s_T
  a_MN_10.state = s_I     a_MN_10.state = s_I     a_MN_10.state = s_R
```

Figura 5.28 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.14

5.2.15 Propriedade 5.15, resultado gerado: contraexemplo

A propriedade 5.15 especifica que não existe um futuro caminho onde globalmente, a partir de cada estado que compõe o caminho, exista um outro caminho onde uma transmissão de um pacote por parte do primeiro nodo da rede implique na existência de um caminho futuro onde o último nodo recebe o que foi transmitido.

Como a propriedade possui uma fórmula mais externa composta pelo operador □ o contraexemplo gerado pelo SAN *Model Checker* é um *trace* que percorre os estados que compõem a fórmula com o operador □ sem criar outros *traces* para a fórmula interna.

O resultado gerado pelo SAN *Model Checker*, Figura 5.29 mostra uma testemunha com 33 estados onde um ciclo é detecado nos estados que estão nas linhas 21 e 39 respectivamente.

O resultado gerado pelo NuSMV Figura 5.30, mostra uma testemunha com 19 estados, onde um ciclo foi detectado no estado onde todos os nodos estão em I, estados 1.1 e 1.19 respectivamente.

```
1 Contra-exemplo gerado pela ferramenta para especificação: ~(EG( (st MN_1 == T) -> EF( (st MN_10 == R))))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 I - I - I - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g12
8 T - R - I - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx1
9 I - R - I - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g23
10 I - T - R - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx2
11 I - I - R - I - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g34
12 I - I - T - R - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx3
13 I - I - I - R - I - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g45
14 I - I - I - T - R - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx4
15 I - I - I - I - R - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g12
16 T - R - I - I - R - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx1
17 I - R - I - I - R - I - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g56
18 I - R - I - I - T - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx5
19 I - R - I - I - I - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g23
20 I - T - R - I - I - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx2
21 I - I - R - I - I - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g67
22 I - I - R - I - I - T - R - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx6
23 I - I - R - I - I - I - R - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g34
24 I - I - T - R - I - I - R - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx3
25 I - I - I - R - I - I - R - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g78
26 I - I - I - R - I - I - T - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx7
27 I - I - I - R - I - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g45
28 I - I - I - T - R - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx4
29 I - I - I - I - R - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g12
30 T - R - I - I - R - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx1
31 I - R - I - I - R - I - I - R - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g89
32 I - R - I - I - R - I - I - T - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx8
33 I - R - I - I - R - I - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g56
34 I - R - I - I - T - R - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx5
35 I - R - I - I - I - R - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g23
36 I - T - R - I - I - R - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx2
37 I - I - R - I - I - R - I - I - R - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - g910
38 I - I - R - I - I - R - I - I - T - R -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) - tx9
39 I - I - R - I - I - R - I - I - I - I -  EG(~( (st MN_1 == T)^~(E(True U  (st MN_10 == R))))) -  inicio do ciclo
```

Figura 5.29 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.15

```
-- specification !(EG (a_MN_1.state = s_T -> EF a_MN_10.state = s_R))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-       -> State: 1.7 <-       -> State: 1.13 <-      -> State: 1.19 <-
  a_MN_1.state = s_I     a_MN_1.state = s_I     a_MN_1.state = s_I     a_MN_1.state = s_I
  a_MN_2.state = s_I     a_MN_2.state = s_I     a_MN_2.state = s_I     a_MN_2.state = s_I
  a_MN_3.state = s_I     a_MN_3.state = s_I     a_MN_3.state = s_I     a_MN_3.state = s_I
  a_MN_4.state = s_I     a_MN_4.state = s_R     a_MN_4.state = s_I     a_MN_4.state = s_I
  a_MN_5.state = s_I     a_MN_5.state = s_I     a_MN_5.state = s_I     a_MN_5.state = s_I
  a_MN_6.state = s_I     a_MN_6.state = s_I     a_MN_6.state = s_I     a_MN_6.state = s_I
  a_MN_7.state = s_I     a_MN_7.state = s_I     a_MN_7.state = s_R     a_MN_7.state = s_I
  a_MN_8.state = s_I     a_MN_8.state = s_I     a_MN_8.state = s_I     a_MN_8.state = s_I
  a_MN_9.state = s_I     a_MN_9.state = s_I     a_MN_9.state = s_I     a_MN_9.state = s_I
  a_MN_10.state = s_I    a_MN_10.state = s_I    a_MN_10.state = s_I    a_MN_10.state = s_I
-> State: 1.2 <-       -> State: 1.8 <-       -> State: 1.14 <-
  a_MN_1.state = s_T     a_MN_1.state = s_I     a_MN_1.state = s_I
  a_MN_2.state = s_R     a_MN_2.state = s_I     a_MN_2.state = s_I
  a_MN_3.state = s_I     a_MN_3.state = s_I     a_MN_3.state = s_I
  a_MN_4.state = s_I     a_MN_4.state = s_T     a_MN_4.state = s_I
  a_MN_5.state = s_I     a_MN_5.state = s_R     a_MN_5.state = s_I
  a_MN_6.state = s_I     a_MN_6.state = s_I     a_MN_6.state = s_I
  a_MN_7.state = s_I     a_MN_7.state = s_I     a_MN_7.state = s_T
  a_MN_8.state = s_I     a_MN_8.state = s_I     a_MN_8.state = s_R
  a_MN_9.state = s_I     a_MN_9.state = s_I     a_MN_9.state = s_I
  a_MN_10.state = s_I    a_MN_10.state = s_I    a_MN_10.state = s_I
-> State: 1.3 <-       -> State: 1.9 <-       -> State: 1.15 <-
  a_MN_1.state = s_I     a_MN_1.state = s_I     a_MN_1.state = s_I
  a_MN_2.state = s_R     a_MN_2.state = s_I     a_MN_2.state = s_I
  a_MN_3.state = s_I     a_MN_3.state = s_I     a_MN_3.state = s_I
  a_MN_4.state = s_I     a_MN_4.state = s_I     a_MN_4.state = s_I
  a_MN_5.state = s_I     a_MN_5.state = s_R     a_MN_5.state = s_I
  a_MN_6.state = s_I     a_MN_6.state = s_I     a_MN_6.state = s_I
  a_MN_7.state = s_I     a_MN_7.state = s_I     a_MN_7.state = s_I
  a_MN_8.state = s_I     a_MN_8.state = s_I     a_MN_8.state = s_R
  a_MN_9.state = s_I     a_MN_9.state = s_I     a_MN_9.state = s_I
  a_MN_10.state = s_I    a_MN_10.state = s_I    a_MN_10.state = s_I
-> State: 1.4 <-       -> State: 1.10 <-      -> State: 1.16 <-
  a_MN_1.state = s_I     a_MN_1.state = s_I     a_MN_1.state = s_I
  a_MN_2.state = s_T     a_MN_2.state = s_I     a_MN_2.state = s_I
  a_MN_3.state = s_R     a_MN_3.state = s_I     a_MN_3.state = s_I
  a_MN_4.state = s_I     a_MN_4.state = s_I     a_MN_4.state = s_I
  a_MN_5.state = s_I     a_MN_5.state = s_T     a_MN_5.state = s_I
  a_MN_6.state = s_I     a_MN_6.state = s_R     a_MN_6.state = s_I
  a_MN_7.state = s_I     a_MN_7.state = s_I     a_MN_7.state = s_I
  a_MN_8.state = s_I     a_MN_8.state = s_I     a_MN_8.state = s_T
  a_MN_9.state = s_I     a_MN_9.state = s_I     a_MN_9.state = s_R
  a_MN_10.state = s_I    a_MN_10.state = s_I    a_MN_10.state = s_I
-> State: 1.5 <-       -> State: 1.11 <-      -> State: 1.17 <-
  a_MN_1.state = s_I     a_MN_1.state = s_I     a_MN_1.state = s_I
  a_MN_2.state = s_I     a_MN_2.state = s_I     a_MN_2.state = s_I
  a_MN_3.state = s_R     a_MN_3.state = s_I     a_MN_3.state = s_I
  a_MN_4.state = s_I     a_MN_4.state = s_I     a_MN_4.state = s_I
  a_MN_5.state = s_I     a_MN_5.state = s_I     a_MN_5.state = s_I
  a_MN_6.state = s_I     a_MN_6.state = s_R     a_MN_6.state = s_I
  a_MN_7.state = s_I     a_MN_7.state = s_I     a_MN_7.state = s_I
  a_MN_8.state = s_I     a_MN_8.state = s_I     a_MN_8.state = s_I
  a_MN_9.state = s_I     a_MN_9.state = s_I     a_MN_9.state = s_R
  a_MN_10.state = s_I    a_MN_10.state = s_I    a_MN_10.state = s_I
-> State: 1.6 <-       -> State: 1.12 <-      -> State: 1.18 <-
  a_MN_1.state = s_I     a_MN_1.state = s_I     a_MN_1.state = s_I
  a_MN_2.state = s_I     a_MN_2.state = s_I     a_MN_2.state = s_I
  a_MN_3.state = s_T     a_MN_3.state = s_I     a_MN_3.state = s_I
  a_MN_4.state = s_R     a_MN_4.state = s_I     a_MN_4.state = s_I
  a_MN_5.state = s_I     a_MN_5.state = s_I     a_MN_5.state = s_I
  a_MN_6.state = s_I     a_MN_6.state = s_T     a_MN_6.state = s_I
  a_MN_7.state = s_I     a_MN_7.state = s_R     a_MN_7.state = s_I
  a_MN_8.state = s_I     a_MN_8.state = s_I     a_MN_8.state = s_I
  a_MN_9.state = s_I     a_MN_9.state = s_I     a_MN_9.state = s_T
  a_MN_10.state = s_I    a_MN_10.state = s_I    a_MN_10.state = s_R
```

Figura 5.30 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.15

5.2.16 Propriedade 5.16, resultado gerado: testemunha

A propriedade 5.16 verifica se existe um caminho futuro onde a última estação esteja bloqueada seja alcançado.

O resultado gerado pelos verificadores foi diferente, inclusive a resposta gerada começa por um estado inicial diferente. Como pode ser visto na Figura 5.32 o NuSMV começa a testemunha pelo estado inicial $(0\_1, 0\_0)$ enquanto o estado inicial pelo qual o SAN *Model Checker*, Figura 5.31, inicia o *trace* é $(0\_0, 0\_0)$. Para chegar a um estado onde a última estação está bloqueada, o SAN *Model Checker* encontrou uma testemunha com oito estados e o estado de bloqueio encontrado foi $(1\_1, 1\_2)$.

```
1 Testemunha gerada pela ferramentapara especificação: E(true U  (st M3 == st_1_2))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 st_0_0 - st_0_0 -  E(True U  (st M3 == st_1_2)) - r1_2
8 st_0_1 - st_0_0 -  True - r1_2
9 st_1_1 - st_0_0 -  True - r1_2
10 st_1_2 - st_0_0 -  True - r2_3
11 st_1_1 - st_0_1 -  True - r1_2
12 st_1_2 - st_0_1 -  True - r2_3
13 st_1_1 - st_1_1 -  True - r2_3_b
14 st_1_1 - st_1_2 -   (st M3 == st_1_2) -  fim do caminho
```

Figura 5.31 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.16

```
-- specification !E [ TRUE U a_M3.state = s_st_1_2 ]   is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_M2.state = s_st_0_1
  a_M3.state = s_st_0_0
-> State: 1.2 <-
  a_M2.state = s_st_1_1
  a_M3.state = s_st_0_0
-> State: 1.3 <-
  a_M2.state = s_st_1_2
  a_M3.state = s_st_0_0
-> State: 1.4 <-
  a_M2.state = s_st_1_1
  a_M3.state = s_st_0_1
-> State: 1.5 <-
  a_M2.state = s_st_0_1
  a_M3.state = s_st_1_1
-> State: 1.6 <-
  a_M2.state = s_st_0_1
  a_M3.state = s_st_1_2
```

Figura 5.32 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.16

5.2.17 Propriedade 5.17, resultado gerado: testemunha

A propriedade 5.17 apresenta o comportamento de *liveness* da última estação.

O resultado gerado pelo SAN *Model Checker* (Figura 5.33), mostra uma testemunha de dois estados, onde a partir de um estado de bloqueio um outro estado em que a última estação é desbloqueada é alcançado.

A testemunha gerada pelo NuSMV (Figura 5.34), contém um estado inicial, no qual a última estação está desbloqueada, comprovando assim que existe um estado atendendo à especificação 5.17. Este tipo de resultado com apenas um estado é o mesmo obtido na propriedade 5.3. Isto mostra que o NuSMV gera contraexemplos ou testemunhas com apenas um estado, desde que o estado que satisfaz ou refuta a fórmula seja um estado inicial do modelo.

```
1 Testemunha gerada pela ferramentapara especificação: EF( (st M3 == st_1_2) -> AF(~( (st M3 == st_1_2))))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 st_0_0 - st_0_0 -  E(True U ~( (st M3 == st_1_2)^EG( (st M3 == st_1_2)))) - r1_2
8 st_0_1 - st_0_0 -  ~( (st M3 == st_1_2)^EG( (st M3 == st_1_2))) -  fim do caminho
```

Figura 5.33 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.17

```
-- specification !(EF (a_M3.state = s_st_1_2 -> AF a_M3.state != s_st_1_2))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_M2.state = s_st_0_1
  a_M3.state = s_st_0_0
```

Figura 5.34 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.17

5.2.18 Propriedade 5.18, resultado gerado: contraexemplo

A propriedade 5.18 apresenta semântica semelhante a propriedade anterior, contudo exige que exista algum caminho onde a partir de um estado de bloqueio a última estação seja desbloqueada.

O resultado gerado pelo SAN *Model Checker* Figura 5.35, devido a semelhança entre as especificações foi o mesmo gerado para a propriedade 5.17. O mesmo é verdade para o resultado obtido pelo NuSMV, Figura 5.36

```
1 Testemunha gerada pela ferramentapara especificação: EF( (st M3 == st_1_2) -> EF(~( (st M3 == st_1_2))))
2
3 ** A ordem em que os estados aparecem e a mesma de declaracao dos automatos no modelo SAN **
4
5  Estado Global -  (sub)formula valida no estado -  evento disparado
6
7 st_0_0 - st_0_0 -  E(True U ~( (st M3 == st_1_2)^~(E(True U ~( (st M3 == st_1_2)))))) - r1_2
8 st_0_1 - st_0_0 -  ~( (st M3 == st_1_2)^~(E(True U ~( (st M3 == st_1_2))))) -  fim do caminho
```

Figura 5.35 – Resultado gerado pelo SAN *Model Checker* para a propriedade 5.18

```
-- specification !(EF (a_M3.state = s_st_1_2 -> EF a_M3.state != s_st_1_2))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_M2.state = s_st_0_1
  a_M3.state = s_st_0_0
```

Figura 5.36 – Resultado gerado pelo NuSMV *Model Checker* para a propriedade 5.18

5.2.19 Demais propriedades testadas no SAN *Model Checker*

Para ampliar o número de fórmulas CTL avaliadas pelo SAN *Model Checker* e verificar mensagens de saída que podem ser exibidas pela ferramenta ao usuário, mais algumas propriedades CTL foram definidas. Elas foram criadas através da combinação de operadores temporais e fórmulas compostas pelos quantificadores universal e existencial. A Tabela 5.1 mostra os resultados obtidos para estas propriedades avaliadas sobre o modelo do Jantar dos Filósofos. Para cada uma delas, foi solicitada a geração de uma testemunha e um contraexemplo.

A Tabela 5.1 mostra que, em alguns casos, a ferramenta encontrou uma testemunha incompleta, ou seja, situações em que o resultado gerado mostra parte da satisfação ou refutação da fórmula. Para outros casos, o verificador identificou um resultado em árvore, não suportado pela ferramenta, e nestes casos mostrou apenas o estado inicial que satisfaz ou refuta a fórmula; situações em que o verificador não encontrou um estado inicial que satisfizesse a fórmula, nesse caso uma testemunha ou contraexemplo não foi gerado, representado pelo símbolo $\times$ e situações onde um resultado completo, isto é, que mostra toda a satisfação ou refutação da fórmula, foi encontrado, representado pelo símbolo $\checkmark$.

Tabela 5.1 – Combinação de operadores temporais avaliados na Ferramenta

ID	Propriedade	Testemunha	Contraexemplo
5.19	$\forall\Diamond\exists\Diamond Fil0 = Thinking$	Em árvore	×
5.20	$\forall\Diamond\exists\Box Fil0 = Thinking$	Em árvore	×
5.21	$\forall\Diamond\exists\bigcirc Fil0 = Thinking$	Em árvore	×
5.22	$\forall\Diamond\exists(Fil0 = Thinking \cup Fil0 = Right)$	Em árvore	×
5.23	$\forall\Box\exists\Diamond Fil0 = Thinking$	Em árvore	×
5.24	$\forall\Box\exists\Box Fil0 = Thinking$	×	✓
5.25	$\forall\Box\exists\bigcirc Fil0 = Thinking$	×	✓
5.26	$\forall\Box\exists(Fil0 = Thinking \cup Fil0 = Right)$	×	✓
5.27	$\forall\bigcirc\exists\Diamond Fil0 = Thinking$	Em árvore	×
5.28	$\forall\bigcirc\exists\Box Fil0 = Thinking$	×	✓
5.29	$\forall\bigcirc\exists\bigcirc Fil0 = Thinking$	×	✓
5.30	$\forall\bigcirc\exists(Fil0 = Thinking \cup Fil0 = Right)$	Em árvore	×
5.31	$\forall(Fil0 = Thinking \cup \exists\Diamond Fil0 = Right)$	Em árvore	×
5.32	$\forall(Fil0 = Thinking \cup \exists\Box Fil0 = Right)$	×	Em árvore
5.33	$\forall(Fil0 = Thinking \cup \exists\bigcirc Fil0 = Right)$	Em árvore	×
5.34	$\exists\Diamond\forall\Diamond Fil0 = Thinking$	Testemunha incompleta	×
5.35	$\exists\Diamond\forall\Box Fil0 = Thinking$	×	Em árvore
5.36	$\exists\Diamond\forall\bigcirc Fil0 = Thinking$	Testemunha incompleta	×
5.37	$\exists\Diamond\forall(Fil0 = Thinking \cup Fil0 = Right)$	Testemunha incompleta	×
5.38	$\exists\Box\forall\Diamond Fil0 = Thinking$	Testemunha incompleta	×
5.39	$\exists\Box\forall\Box Fil0 = Thinking$	×	Em árvore
5.40	$\exists\Box\forall\bigcirc Fil0 = Thinking$	×	Em árvore
5.41	$\exists\Box\forall(Fil0 = Thinking \cup Fil0 = Right)$	×	Em árvore
5.42	$\exists\bigcirc\forall\Diamond Fil0 = Thinking$	Testemunha incompleta	×
5.43	$\exists\bigcirc\forall\Box Fil0 = Thinking$	×	Em árvore
5.44	$\exists\bigcirc\forall\bigcirc Fil0 = Thinking$	×	Em árvore
5.45	$\exists\bigcirc\forall(Fil0 = Thinking \cup Fil0 = Right)$	Testemunha incompleta	×
5.46	$\exists(Fil0 = Thinking \cup \forall\Diamond Fil0 = Right)$	×	Em árvore
5.47	$\exists(Fil0 = Thinking \cup \forall\bigcirc Fil0 = Right)$	Testemunha incompleta	×
5.48	$\exists(Fil0 = Thinking \cup \forall\Box Fil0 = Right)$	×	Em árvore

Conforme a Tabela 5.1 mostra, em cinco resultados (5.24, 5.25, 5.26, 5.28 e 5.29), o SAN *Model Checker* chegou a um resultado que explicava, por completo, a refutação de uma especificação pelo modelo. Nas demais propriedades testadas, o verificador chegou a resultados em árvore, incompletos ou, ainda, não houve saída para o usuário, uma vez que, não existiam estados iniciais do modelo que fizessem parte do resultado. Para estes casos, o verificador exibe uma mensagem ao usuário, informando o tipo de resultado (incompleto ou em árvore) gerada, ou ainda, de que não foi possível gerá-lo.

5.3 Tempo e memória

Esta seção mostra o tempo e memória utilizados para gerar os resultados apresentados pela Seção 5.2. A Tabela 5.2 mostra essas informações da seguinte forma: a coluna *Propriedade* mostra a qual propriedade as informações de tempo e memória se referem; a coluna *Verificação* mostra os o tempo (em segundos) e memória (em megabytes) necessários para a verificação da especificação no modelo SAN. Torna-se importante destacar que a coluna *Tempo* sob o campo *Verificação* mostra o tempo total incluindo todos os processos envolvidos na verificação (geração do espaço de estados atingível, transformação de uma fórmula CTL em ENF e eventualmente o tratamento dela, para o caso da geração de um contraexemplo, função de rotulação de estados e cálculo do conjunto de satisfação de fórmulas - SAT). A coluna *Memória* sob o campo *Verificação* mostra a memória necessária para armazenar no formato de diagrama de decisão multivalorado (MDD) o conjunto de estados que satisfazem a fórmula avaliada. As colunas *Tempo* e *Memória* sob o campo *Contraexemplo/testemunha* mostram o tempo e memória utilizados para a geração de um contraexemplo ou de uma testemunha.

Tabela 5.2 – Tempo e memória consumidos pelo SAN *Model Checker*.

Propriedade	Verificação		Contraexemplo/testemunha	
	Tempo (s)	Memória (MB)	Tempo (s)	Memória (MB)
5.1	2,45 x 10^{-4}	0,019	3,60 x 10^{-5}	0,003
5.2	1,89 x 10^{-4}	0,022	5,20 x 10^{-5}	0,003
5.3	1,16 x 10^{-4}	0,012	3,10 x 10^{-5}	0,005
5.4	1,46 x 10^{-4}	0,016	3,10 x 10^{-5}	0,003
5.5	2,46 x 10^{-4}	0,025	3,00 x 10^{-5}	0,003
5.6	2,60 x 10^{-4}	0,026	2,10 x 10^{-5}	0,003
5.7	1,72 x 10^{-4}	0,018	2,19 x 10^{-5}	0,003
5.8	1,47 x 10^{-4}	0,019	3,79 x 10^{-5}	0,003
5.9	2,35 x 10^{-4}	0,021	5,01 x 10^{-5}	0,003
5.10	1,61 x 10^{-4}	0,024	1,69 x 10^{-5}	0,003
5.11	2,21 x 10^{-4}	0,018	2,10 x 10^{-5}	0,003
5.12	6,60 x 10^{-3}	0,315	3,85 x 10^{-4}	0,010
5.13	5,37 x 10^{-3}	0,302	2,33 x 10^{-4}	0,014
5.14	1,23 x 10^{-2}	0,434	3,84 x 10^{-4}	0,010
5.15	6,53 x 10^{-3}	0,315	3,96 x 10^{-4}	0,010
5.16	2,45 x 10^{-4}	0,018	4,32 x 10^{-5}	0,002
5.17	8,20 x 10^{-5}	0,007	2,29 x 10^{-5}	0,002
5.18	5,70 x 10^{-5}	0,006	2,31 x 10^{-5}	0,002

A Tabela 5.3 mostra o consumo de tempo e memória verificados para as propriedades apresentadas na Tabela 5.1.

O consumo de memória para geração de contraexemplos e testemunhas apresentado foi baixo, como pode ser visto nas Tabelas 5.2 e 5.3. Isso ocorreu pelo fato de que os modelos são pequenos, mas principalmente pelo fato de que a estrutura MDD, que codifica os estados que

Tabela 5.3 – Consumo de memória e tempo para Tabela 5.2

Propriedade	Verificação		Contraexemplo/testemunha	
	Tempo (s)	Memória (MB)	Tempo (s)	Memória (MB)
5.19	2,42 x 10^{-4}	0,014	1.72 x 10^{-5}	0,002
5.20	3,59 x 10^{-4}	0,019	1,38 x 10^{-5}	0,002
5.21	2,78 x 10^{-4}	0,017	1,50 x 10^{-5}	0,002
5.22	2,93 x 10^{-4}	0,016	1,69 x 10^{-5}	0,002
5.23	3,83 x 10^{-4}	0,015	1,60 x 10^{-5}	0,002
5.24	2,32 x 10^{-4}	0,014	3,91 x 10^{-5}	0,003
5.25	3,09 x 10^{-4}	0,018	4,10 x 10^{-5}	0,003
5.26	3,24 x 10^{-4}	0,018	9,18 x 10^{-5}	0,003
5.27	2,94 x 10^{-4}	0,015	1,41 x 10^{-5}	0,002
5.28	2,50 x 10^{-4}	0,016	3,00 x 10^{-5}	0,002
5.29	3,99 x 10^{-4}	0,017	3,81 x 10^{-5}	0,002
5.30	6,14 x 10^{-4}	0,017	1,50 x 10^{-5}	0,002
5.31	4,58 x 10^{-4}	0,015	2,10 x 10^{-5}	0,002
5.32	5,20 x 10^{-4}	0,021	1,50 x 10^{-5}	0,002
5.33	1,43 x 10^{-3}	0,020	2,22 x 10^{-5}	0,002
5.34	4,45 x 10^{-4}	0,019	4,58 x 10^{-5}	0,003
5.35	2,57 x 10^{-4}	0,014	1,41 x 10^{-5}	0,002
5.36	3,68 x 10^{-3}	0,023	1,87 x 10^{-4}	0,004
5.37	3,33 x 10^{-4}	0,018	4,51 x 10^{-5}	0,003
5.38	4,12 x 10^{-4}	0,019	5,10 x 10^{-5}	0,002
5.39	1,73 x 10^{-4}	0,013	1,31 x 10^{-5}	0,002
5.40	2,79 x 10^{-4}	0,018	1,50 x 10^{-5}	0,002
5.41	3,11 x 10^{-4}	0,017	1,41 x 10^{-5}	0,002
5.42	6,33 x 10^{-4}	0,022	3.91 x 10^{-5}	0,002
5.43	2,72 x 10^{-4}	0,014	1.60 x 10^{-5}	0,002
5.44	3,72 x 10^{-4}	0,020	1.60 x 10^{-5}	0,002
5.45	3.72 x 10^{-4}	0,018	4.20 x 10^{-5}	0,002
5.46	2.85 x 10^{-4}	0,015	4.10 x 10^{-5}	0,003
5.47	4.10 x 10^{-4}	0,023	5.79 x 10^{-5}	0,003
5.48	1.11 x 10^{-4}	0,013	7.15 x 10^{-6}	0,002

satifazem a fórmula avaliada após a verificação, ser mantida para a geração de contraexemplos e testemunhas. Dessa forma, o pequeno consumo de memória verificado foi resultado de operações de conjunto, necessárias durante o cálculo de uma testemunha ou contraexemplo, realizadas sobre o espaço de estados rotulado com cada (sub)fórmula e codificado em forma de MDD. O tempo necessário para a geração dos resultados também foi baixo, uma vez que, com o espaço de estados rotulado, o gerador de testemunha visita apenas os estados que podem compor um contraexemplo ou testemunha, simplificando a busca.

6. CONSIDERAÇÕES FINAIS

O trabalho apresentado ao longo desta dissertação está inserido dentro do projeto de construção de um verificador de modelos para o formalismo de Redes de Autômatos Estocásticos, verificador que recebeu o nome de SAN *Model Checker*, e teve como objetivo implementar a geração de contraexemplos e testemunhas para a ferramenta. Para informações sobre os demais trabalhos que compõem o projeto, recomenda-se a leitura do relatório técnico do aluno de mestrado Alberto Wondraceck [34] e a dissertação de mestrado de Lucas Oleksinski [27]. Salienta-se que, além destes trabalhos, o aluno de mestrado Eli Maruani contribuiu no projeto de pesquisa.

Um verificador de modelos pode simplesmente informar ao usuário se um modelo atende ou não a uma especificação avaliada. Contudo, a implementação de contraexemplos e testemunhas em um verificador torna-se relevante quando considerado o valor que um contraexemplo tem para depuração do sistema. O modelador, após a avaliação de uma especificação pelo verificador, de posse do contraexemplo gerado pela ferramenta, pode localizar com maior facilidade o erro que levou o modelo a refutar a especificação avaliada. O Verificador também pode gerar uma testemunha, este resultado ratifica que o modelo atende a uma especificação.

A estratégia implementada para a geração dos resultados foi a de armazenar os valores que codificam os estados que satisfazem cada subfórmula da fórmula avaliada à medida que o algoritmo de satisfação de fórmulas CTL é executado. Como consequência dessa abordagem, a geração dos resultados é feita após o término do algoritmo de satisfação de fórmulas. Essa abordagem é semelhante à empregada por Keijo Heljanko [21].

Conforme abordado no Capítulo 2, contraexemplos são gerados para fórmulas quantificadas universalmente, enquanto que testemunhas são geradas para fórmulas quantificadas existencialmente. Apesar disso, a opção tomada foi a de dar liberdade para o usuário escolher o tipo de saída que deseja obter, isto é, o usuário pode entrar com uma especificação quantificada existencialmente e solicitar um contraexemplo ou ainda entrar com uma especificação quantificada universalmente e solicitar uma testemunha. Cabe à ferramenta tratar adequadamente estas situações.

É importante colocar que, apesar de existir um cuidado em implementar da melhor forma possível a geração de contraexemplos e testemunhas, a busca por uma implementação que resultasse no melhor desempenho que se pudesse obter no que se refere a tempo de geração de resultados e consumo de memória esteve, desde o começo, fora do escopo deste trabalho, que se deteve à geração de resultados coerentes com os conceitos de contraexemplos e testemunhas descritos na literatura de *Model Checking*. Contudo, destaca-se que os resultados de tempo e memória obtidos para a geração dos resultados mostraram-se satisfatórios, uma vez que pouca memória e tempo foram despendidos pela ferramenta durante o cálculo do contraexemplo ou testemunha. Mesmo que para uma mesma especificação possa existir mais de um contraexemplo ou testemunha e que por esse motivo chegar a uma mesma resposta em ambos os verificadores seria difícil, os resultados obtidos pelo SAN *Model Checker* foram comparados aos resultados gerados pelo NuSMV *Model Checker*. Esta comparação mostrou que em seis propriedades ambos os verificadores chegaram

à mesma resposta, dando um forte indício do funcionamento correto da implementação realizada neste trabalho. Em outros resultados o NuSMV exibiu contraexemplos e testemunhas de apenas um estado, enquanto que os resultados gerados pelo SAN *Model Checker* possuem no mínimo dois estados, quando os resultados não são em formato de árvore, como visto no Capítulo 5.

A contribuição desta dissertação está no fato de que o SAN *Model Checker*, implementado ao longo do mestrado, agora conta com o recurso de geração de contraexemplos e testemunhas para o usuário, além de que a lacuna de inexistência de um verificador para Redes de Autômatos Estocásticos ter sido preenchida pela ferramenta.

Durante a implementação, questões sobre a forma de gerar os resultados surgiram e elas podem guiar trabalhos futuros quanto à geração de contraexemplos e testemunhas, na intenção de aprimorar a técnica empregada ou, ainda, criar uma alternativa a ela. Por exemplo, a estratégia adotada para geração de contraexemplos e testemunhas foi a de gerar os resultados após a execução do algoritmo de satisfação de fórmulas CTL. Contudo, conforme Clarke [12] isto não é necessário, e a geração pode ser realizada ao mesmo tempo em que o algoritmo de satisfação de fórmulas é executado. Diante disso, fica em aberto a possibilidade de implementar uma estratégia que faça uso dessa possibilidade.

Outras possibilidades consideradas são as de empregar técnicas diferentes para geração nos trabalhos desenvolvidos em paralelo no projeto. A saber: uma abordagem que faça uso de processamento paralelo e distribuído para a geração de contraexemplos e testemunhas e que poderia ser utilizada em conjunto com o trabalho apresentado em [27]. Como também uma estratégia para a geração de contraexemplos e testemunhas que seja adequada ao trabalho de implementação da saturação no verificador, trabalho atualmente em desenvolvimento no projeto.

A implementação realizada neste trabalho gera contraexemplos e testemunhas lineares, ou seja, *traces* de execução do sistema. Desse modo, também consta no conjunto de trabalhos futuros a geração de contraexemplos e testemunhas em formato de árvore, visto que, esse tipo de resultado abrange um maior número de especificações em CTL, e gera um resultado mais completo para a satisfação ou refutação da propriedade avaliada [13, 14], quando ela tem formato em árvore.

Dito isto, esta dissertação representa os primeiros passos na criação do verificador de modelos descritos em SAN, além de criar a base e lançar possibilidades de trabalhos futuros para a geração de contraexemplos e testemunhas para ferramenta.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. "Basic concepts and taxonomy of dependable and secure computing", *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, jan-mar 2004, pp. 11–33.

[2] Baier, C.; Katoen, J.-P. "Principles of Model Checking". MIT Press, 2008, 975p.

[3] Balbo, G. "Introduction to stochastic Petri nets", *Lecture Notes in Computer Science*, vol. 2090, 2001, pp. 84–155.

[4] Baldo, L.; Brenner, L.; Fernandes, L. G.; Fernandes, P.; Sales, A. "Performance Models for Master/Slave Parallel Programs", *Electronic Notes In Theoretical Computer Science (ENTCS)*, vol. 128–4, 2005, pp. 101–121.

[5] Brenner, L.; Fernandes, P.; Sales, A. "The Need for and the Advantages of Generalized Tensor Algebra for Structured Kronecker Representations", *International Journal of Simulation: Systems, Science & Technology (IJSIM)*, vol. 6, 2005, pp. 52–60.

[6] Broadfoot, P. J.; Roscoe, A. W. "Tutorial on FDR and its applications". In: SPIN, Havelund, K.; Penix, J.; Visser, W. (Editores), 2000, pp. 322.

[7] Buccafurri, F.; Eiter, T.; Gottlob, G.; Leone, N. "On ACTL formulas having linear counterexamples", *J. Comput. Syst. Sci*, vol. 62–3, 2001, pp. 463–515.

[8] Ciardo, G.; Miner, A. S. "SMART: The stochastic model checking analyzer for reliability and timing". In: International Conference on the Quantitative Evaluation of Systems, Enschede, The Netherlands, 2004, pp. 338–339.

[9] Cimatti, A.; Clarke, E. M.; Giunchiglia, F.; Roveri, M. "NUSMV: A new symbolic model checker", *International Journal on Software Tools for Technology Transfer - STTT*, vol. 2–4, 2000, pp. 410–425.

[10] Clarke, E. M. "The birth of model checking". In: 25 Years of Model Checking, Grumberg, O.; Veith, H. (Editores), 2008, pp. 1–26.

[11] Clarke, E. M.; Grumberg, O.; McMillan, K. L.; Zhao, X. "Efficient generation of counterexamples and witnesses in symbolic model checking". In: Design Automation Conference - DAC, 1995, pp. 427–432.

[12] Clarke, E. M.; Grumberg, O.; Peled, A. D. "Model Checking". MIT Press, Cambridge, Massachusetts, 1999, 330p.

[13] Clarke, E. M.; Jha, S.; Lu, Y.; Veith, H. "Tree-like counterexamples in model checking". In: Logic in Computer Science, 2002, pp. 19–29.

[14] Clarke, E. M.; Veith, H. "Counterexamples revisited: Principles, algorithms, applications". In: Verification: Theory and Practice, Dershowitz, N. (Editor), 2003, pp. 208–224.

[15] Correa, C.; Dotti, F.; Fernandes, P.; Maruani, E.; Oleksinski, L.; Sales, A. "Um verificador de modelos descritos em Redes de Autômatos Estocásticos". In: SBRC 2012 - WTF (Workshop de Testes e Tolerancia a Falhas), 2012, pp. 115–128.

[16] Czekster, R. "Solução numérica de descritores Markovianos a partir de re-estruturações de termos tensoriais", Tese de Doutorado, Pontifícia Universidade Católica do Rio Grande do Sul, Brasil, 2010, 194p.

[17] Dotti, F. L.; Fernandes, P.; Sales, A.; Santos, O. M. "Modular Analytical Performance Models for Ad Hoc Wireless Networks". In: Proceedings of the 3rd International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt 2005), Trentino, Italy, 2005, pp. 164–173.

[18] E.M. Clarke; E.A. Emerson; A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM Transactions on Programming Languages and Systems*, vol. 8–2, Abr 1986, pp. 244–263.

[19] Fernandes, P.; O'Kelly, M.; Papadopoulos, C.; Sales, A. "Analysis of exponential reliable production lines using kronecker descriptors", *International Journal of Production Research*, –ahead-of-print, 2013, pp. 1–18.

[20] Fernandes, P.; Sales, A.; Santos, A.; Webber, T. "Performance Evaluation of Software Development Teams: a Practical Case Study", *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 275, 2011, pp. 73–92.

[21] Heljanko, K. "Implementing a ctl model checker". In: In: Workshop Concurrency, Specication Programming. HumboldtUniversitt zu Berlin, Institut fr Informatik, InformatikBericht Nr. 69, 1996, pp. 75–84.

[22] Hinton, A.; Kwiatkowska, M.; Norman, G.; Parker, D. "PRISM: A tool for automatic verification of probabilistic systems". In: Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), Hermanns, H.; Palsberg, J. (Editores), 2006, pp. 441–444.

[23] Hurth, M.; Ryan, M. "Logic In Computer Science". Cambridge University Press, 2004, 427p.

[24] L. Oleksinski, C. Correa, F. D.; Sales, A. "A CTL Model Checker for Stochastic Automata Networks". In: 10th International Conference on the Quantitative Evaluation of Systems (QEST 2013), 2013, pp. 265–268.

[25] Lamport, L. "Proving the correctness of multiprocess programs", *IEEE Transactions on Software Engineering*, vol. 3–2, 1977, pp. 125–143.

[26] Manna, Z.; Pnueli, A. "The temporal logic of reactive and concurrent systems". New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[27] Oleksinski, L. G. "Abordagens paralelas para model checking de SAN", Mestrado, FACIN, Faculdade de Informática, PUCRS, 2013, 123p.

[28] Plateau, B. "On the stochastic structure of parallelism and synchronization models for distributed algorithms". In: SIGMETRICS, 1985, pp. 147–154.

[29] Sales, A. "Réseaux d'Automates Stochastiques: Génération de l'espace d'états atteignables et Multiplication vecteur-descripteur pour une sémantique en temps discret", Tese de Doutorado, Institut Polytechnique de Grenoble, France, 2009, 250p.

[30] Sales, A. "SAN lite-solver: a user-friendly software tool to solve SAN models". In: Spring Simulation Multi-conference (SpringSim'12): SCS/ACM Theory of Modeling and Simulation: DEVS Integrative MS Symposium (DEVS 2012), 2012, pp. 9–16.

[31] Sales, A.; Plateau, B. "Reachable state space generation for structured models which use functional transitions". In: 6th International Conference on the Quantitative Evaluation of Systems (QEST'09), Budapest, Hungary, 2009, pp. 269–278.

[32] Stewart, W. J. "Introduction to the numerical solution of Markov Chains". Princeton University Press, 1994, 539p.

[33] Vergauwen, B.; Lewi, J. "A linear local model checking algorithm for CTL", *Lecture Notes in Computer Science*, vol. 715, 1993, pp. 447–461.

[34] Wondracek, A.; Dotti, F. "Tradução de Modelos Redes de Autômatos Estocásticos para a Linguagem do NuSMV (Translation of SAN models to NuSMV language)", Relatório Técnico 72, PPGCC-FACIN-PUCRS, Porto Alegre, 2013, 15p.

APÊNDICE A – ARQUIVO SAN PARA O PROBLEMA DO JANTAR DOS FILÓSOFOS PARA TRÊS FILÓSOFOS

Modelo SAN em formato textual para o problema do Jantar dos Filósofos apresentado pela Figura 2.1 da Seção 2.1.

```
identifiers

    P = 3; //Número de filósofos
    lambda = 1.000000; //Taxa para aquisição de garfos
    mu = 2.000000; //Taxa de liberação de garfos

    f_1 = (st Fil1 != Comendo) * lambda;
    f_2 = (st Fil2 != Comendo) * lambda;

    f_4 = (st Fil2 == Pensando) * lambda;
    f_5 = (st Fil0 == Pensando) * lambda;

    f_7 = (st Fil1 == Pensando) * lambda;
    f_8 = (st Fil0 != Comendo) * lambda;

events
    loc t_r_0 (f_1);
    loc r_l_0 (f_2);
    loc l_t_0 (mu);

    loc t_r_1 (f_4);
    loc r_l_1 (f_5);
    loc l_t_1 (mu);

    loc t_l_2 (f_7);
    loc l_r_2 (f_8);
    loc r_t_2 (mu);

partial reachability = ((nb Thinking) == P);

network Philosophers (continuous)
    aut Fil2
        stt Thinking to (Left) t_l_2
        stt Left to (Right) l_r_2
        stt Right to (Thinking) r_t_2

    aut Fil1
        stt Thinking to (Right) t_r_1
        stt Right to (Left) r_l_1
        stt Left to (Thinking) l_t_1

    aut Fil0
        stt Thinking to (Right) t_r_0
        stt Right to (Left) r_l_0
        stt Left to (Thinking) l_t_0

    results
        Fil0Thinking = (st Fil0 == Thinking);
```

Figura A.1 – Modelo SAN do Jantar dos Filósofos para três filósofos apresentado em formato textual.

APÊNDICE B – ARQUIVO SAN PARA O MODELO DE REDE WIRELESS ADHOC COM SEIS NODOS

Modelo SAN em formato textual para a rede Adhoc para seis nodos apresentado pela Figura 2.2 da Seção 2.1.6.

identifiers

bandwidth$_b$ = 2000000; //$bandwidthinbits/seconds$

bandwidth$_B$ = $bandwidth_b/8$; //$bandwidthinbytes/seconds$

bandwidth$_M bps$ = $bandwidth_b/1000000$; //$bandwidthinbytes/seconds$

size$_p ack$ = 1500; //$packagesizeinbytes$

RTS = 40; // Request To Send header in bytes

CTS$_A CK$ = 39; //$ClearToSendandAcknowledgeheadersinbytes$

MAC = 47; // Medium Access Control header in bytes

IFT = 50+(3*10)+280; // InterFrame Time (DIFS + 3*SIFS + Average Backoff Time) in microseconds (approximated)

IFS = ((bandwidth$_B$/1000000) $*$ IFT); //$InterFrameSizecostinbytes(approximated)$

overhead = RTS+CTS$_A CK + MAC + IFS$; //$headers$

mu = (bandwidth$_B$)/($size_p ack + overhead$); //$maximumthroughputrate(theoreticallyasmanypackageasthemediumcanhandle)$

lambda = 50000; // package generation rate (one package/DIFS)

f12 = lambda * ((st MN$_3$ == I)($stMN_4$ == I));
f23 = lambda * ((st MN$_1$ == I)($stMN_4$ == I)($stMN_5$ == I));
f34 = lambda * ((st MN$_1$ == I)($stMN_2$ == I)($stMN_5$ == I)($stMN_6$ == I));
f45 = lambda * ((st MN$_2$ == I)($stMN_3$ == I)($stMN_6$ == I));
f56 = lambda * ((st MN$_3$ == I)($stMN_4$ == I));

events

loc tx1 mu; // transmission of one package
loc tx2 mu; // transmission of one package
loc tx3 mu; // transmission of one package
loc tx4 mu; // transmission of one package
syn tx5 mu; // transmission of one package

syn g12 f12; // package generated from 1 to 2
syn g23 f23; // package routed from 2 to 3
syn g34 f34; // package routed from 3 to 4
syn g45 f45; // package routed from 4 to 5
syn g56 f56; // package routed from 5 to 6

partial reachability = (nb I == 6); // initial state

network Ad6 (*continuous*)
$autMN_1$
$sttIto(T)g12$
$sttTto(I)tx1$

$autMN_2$
$sttIto(R)g12$
$sttRto(T)g23$
$sttTto(I)tx2$

$autMN_3$
$sttIto(R)g23$
$sttRto(T)g34$
$sttTto(I)tx3$

$autMN_4$
$sttIto(R)g34$
$sttRto(T)g45$
$sttTto(I)tx4$

$autMN_5$
$sttIto(R)g45$
$sttRto(T)g56$
$sttTto(I)tx5$

$autMN_6$
$sttIto(R)g56$
$sttRto(I)tx5$

results
$t_M N_{1\,I} = stMN_1 == I$;

Figura B.1 – Modelo Adhoc com seis nodos.

APÊNDICE C – ARQUIVO SAN PARA O MODELO LINHA DE PRODUÇÃO COM TRÊS ESTAÇÕES

Arquivo SAN em formato textual para o modelo de Linha de Produção apresentado pela Figura 2.3 da Seção da Seção 2.1.7.

identifiers

$mu_1 = 1.000000$; // taxa de serviço
$mu_2 = 1.000000$; // taxa de serviço
$mu_3 = 1.000000$; // taxa de serviço

// Função de bloqueio (permite que a estação "j" vá para o estado "bloqueando" com taxa "mu_i", indicando que a estação "i" é bloqueada)
$f2\_3\_b = (st\ M2\ != st\_0\_0)$ * mu_2;

events

loc $r1\_2$ (mu_1);
syn $r2\_3$ (mu_2);
loc $r2\_3\_b$ $(f2\_3\_b)$;
loc $r3\_x$ (mu_3);
syn $r3\_x\_u$ (mu_3);

partial reachability = $!((st\ M2 == st\_0\_0)\ \&\&\ (st\ M3 == st\_1\_2))$;

network P_LINE (*continuous*)

$aut\ M2$
$stt\ st\_0\_0\ to\ (st\_0\_1)\ r1\_2$
$stt\ st\_0\_1\ to\ (st\_1\_1)\ r1\_2$
$to\ (st\_0\_0)\ r2\_3\ r3\_x\_u$
$stt\ st\_1\_1\ to\ (st\_1\_2)\ r1\_2$
$to\ (st\_0\_1)\ r2\_3\ r3\_x\_u$
$stt\ st\_1\_2\ to\ (st\_1\_1)\ r2\_3\ r3\_x\_u$

$aut\ M3$
$stt\ st\_0\_0\ to\ (st\_0\_1)\ r2\_3$
$stt\ st\_0\_1\ to\ (st\_1\_1)\ r2\_3$
$to\ (st\_0\_0)\ r3\_x$
$stt\ st\_1\_1\ to\ (st\_1\_2)\ r2\_3\_b$
$to\ (st\_0\_1)\ r3\_x$
$stt\ st\_1\_2\ to\ (st\_1\_1)\ r3\_x\_u$

results

$station\_3\_blocked = (st\ M3 == st\_1\_2)$;

Figura C.1 – Linha de Produção com três estações modelada em SAN descrita em formato textual.