**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ**
**DEPARTAMENTO ACADÊMICO DE INFORMÁTICA**
**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**MATHEUS PEREIRA JÚNIOR**

# UTILIZAÇÃO DE *MODEL CHECKING* NA VERIFICAÇÃO FORMAL DE UM PROTOCOLO DE REDE *FULL DUPLEX* SEM FIO

**TRABALHO DE CONCLUSÃO DE CURSO**

**PONTA GROSSA**
**2016**

**MATHEUS PEREIRA JÚNIOR**

# UTILIZAÇÃO DE *MODEL CHECKING* NA VERIFICAÇÃO FORMAL DE UM PROTOCOLO DE REDE *FULL DUPLEX* SEM FIO

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação do Departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Gleifer Vaz Alves

**PONTA GROSSA**

**2016**

Ministério da Educação
**Universidade Tecnológica Federal do Paraná**
Câmpus Ponta Grossa

Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

## TERMO DE APROVAÇÃO

UTILIZAÇÃO DE MODEL CHECKING NA VERIFICAÇÃO FORMAL DE UM PROTOCOLO DE REDE SEM FIO

por

MATHEUS PEREIRA JÚNIOR

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 12 de Maio de 2016 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Dr. Gleifer Vaz Alves
Orientador

Dr. André Pinz Borges
Membro titular

Dra. Sheila Morais de Almeida
Membro titular

Prof. Dr. Augusto Foronda
Responsável pelo Trabalho de Conclusão de Curso

Prof. Dr. Erikson Freitas de Morais
Coordenador do curso

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

**AGRADECIMENTOS**

Agradeço à todos aqueles que, de alguma maneira, direta ou indiretamente, compartilham desta importante conquista.

Aos professores que instruíram-me durante o curso ensinando os conceitos, fundamentos e técnicas da Computação.

Aos amigos e companheiros de classes que certamente auxiliaram-me durante esta longa caminhada, seja por uma ajuda durante um trabalho, seja por uma simples conversa.

Ao Google (sério) por ser meu tutor sempre ajudando-me na resolução dos problemas sempre presentes na Computação.

Aos meus pais pelo apoio e ajuda durante todo este período.

À Deus que, por sua abundante graça e misericórdia, concedeu-me a vida e a oportunidade de realizar o presente trabalho.

**RESUMO**

JÚNIOR, M.P. Utilização de *Model Checking* na Verificação Formal de um Protocolo de Rede *Full Duplex* sem Fio. 93 p. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2016.

Assegurar o correto funcionamento de sistemas é uma tarefa árdua. Métodos formais têm sido utilizados para tal finalidade. *Model checking* destaca-se como um método formal baseado em modelos utilizado por ter vantagens como sólida fundação matemática e lógica e geração de contra-exemplos, apesar de apresentar o problema da explosão de estados. Modelos são baseados em formalismos como sistemas de transição e cadeias de Markov. Dentre esses autômatos temporais, por apresentarem uma modelagem natural de aspectos temporais, são empregados para modelagem de sistemas em tempo real e protocolos de comunicação. As propriedades de um modelo são formalmente especificadas através de lógicas temporais como LTL e CTL. Exemplos de propriedade incluem ausência de *deadlock* e execução garantida de dadas funcionalidades. *Model checkers* são programas que aplicam o *model checking*. Exemplos incluem PRISM, SPIN e UPPAAL. No presente trabalho um protocolo de controle de acesso ao meio para redes *full duplex* sem fio é modelado através de autômato temporal e formalmente verificado usando o *model checker* UPPAAL. Propriedades como a ocorrência de uma transmissão *full duplex* completada com sucesso e envio de ACK para cada transmissão realizada foram verificadas.

**Palavras-chaves**: Verificação formal. *Model checking*. UPPAAL. 802.11 IEEE. Protocolo MAC *full duplex*.

## ABSTRACT

JÚNIOR, M.P. *Using Model Checking to Formally Verify a Full Duplex MAC Protocol*. 93 p. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Federal University of Technology – Paraná. Ponta Grossa, 2016.

Ensuring the correct behavior of systems is no trivial task. Formal methods have been used in industrial and scientific applications to ensure proper working. Model checking is a model-based formal method that recently has grown in popularity due to its considerable advantages such as solid mathematical-logical foundation and counter-example generation, though it suffers from the state-explosion problem. Models are based on formalisms such as transition systems and Markov chains. Among these, timed automata is widely used to model real time systems and communication protocols since it has a natural way of expressing continuous temporal behavior. Properties of a model are formally specified through temporal logics including LTL and CTL. Examples of properties include deadlock freedom e guaranted function execution of given functionalities. Model checkers are tools that apply the model checking process. SPIN, PRISM e UPPAAL are examples. In this work a full duplex MAC protocol is modeled and verified using UPPAAL. Properties like successful completion of full duplex transmissions and an ACK sent for each successfully completed transmission are verified.

**Keywords**: Formal verification. Model checking. UPPAAL. 802.11 IEEE. Full duplex MAC protocol.

**LISTA DE FIGURAS**


Figura 1 – Ilustração do processo de *model checking* .... 16
Figura 2 – Sistemas de transição de um sistema de cruzamento ferroviário .... 18
Figura 3 – Sistema de transição parcial de um sistema de cruzamento ferroviário .... 19
Figura 4 – Representação visual dos operadores da LTL .... 22
Figura 5 – Representação visual dos operadores da CTL .... 24
Figura 6 – Sistema de transição para uma fórmula CTL não expressível em LTL .... 25
Figura 7 – Sistema de transição para uma fórmula LTL não expressível em CTL .... 26
Figura 8 – Sistema de transição para uma fórmula expressível em LTL e CTL .... 26
Figura 9 – Autômato temporal simples .... 28
Figura 10 – Estrutura interna do SPIN .... 30
Figura 11 – Linguagem PRISM e PTA .... 32
Figura 12 – Visão geral do *model checking* no PRISM .... 32
Figura 13 – Aba de edição de modelos no UPPAAL .... 34
Figura 14 – Aba de especificação de propriedades no UPPAAL .... 34
Figura 15 – Aba de simulação de modelos no UPPAAL .... 35
Figura 16 – Aba de declaração de variáveis, constantes e canais no UPPAAL .... 35
Figura 17 – Modelo de um simples sistema ferroviário no UPPAAL .... 37
Figura 18 – Processo de *backoff* .... 42
Figura 19 – Mecanismo RTS/CTS .... 44
Figura 20 – Trabalhos relacionados ao CSMA/CA HD e FD .... 53
Figura 21 – Passagem de valores síncrona .... 55
Figura 22 – *TIME* unidades de tempo em uma *location* .... 55
Figura 23 – Relação dos componentes do modelo .... 57
Figura 24 – Envio de mensagens .... 59
Figura 25 – Processo de *backoff* .... 60
Figura 26 – Processo de ACK .... 63
Figura 27 – Tratamento de colisão .... 65
Figura 28 – Modelo completo .... 66
Figura 29 – Modelo simplificado (sem *primary timer* e ACKTimeout) .... 68
Figura 30 – Geradores de números aleatórios .... 71
Figura 31 – Verificação da propriedade de *deadlock* .... 75
Figura 32 – Verificação da propriedade nenhuma transmissão em $T_n$ .... 75
Figura 33 – Verificação da propriedade *k* transmissões em $T_n$ .... 76
Figura 34 – Verificação da propriedade de transmissões simultâneas .... 77
Figura 35 – Verificação da propriedade de transmissões simultâneas terminam simultaneamente .... 77
Figura 36 – Verificação da propriedade de ACK na transmissão primária .... 78
Figura 37 – Verificação da propriedade de ACK na transmissão secundária .... 79
Figura 38 – Verificação da propriedade de ACK simultâneo .... 79
Figura 39 – Modelo completo .... 93

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ACK | *Acknowledgment Package* |
| AMCLM | *Adaptive Multi-Services Cross-Layer MAC Protocol* |
| ARQ | *Automatic Repeat Request* |
| CF | *Contra Flow* |
| CSMA/CA | *Carrier Sense Multiple Access/Collision Avoidance* |
| CTL | *Computation Tree Logic* |
| CTS | *Clear to Send* |
| CW | *Contention Window* |
| DCF | *Distribution Coordination Function* |
| DIFS | *DCF Interframe Space* |
| FSM | *Finite State Machine* |
| GUI | *Graphical User Interface* |
| ICT | *Information and Communications Technology* |
| IEEE | *Institute of Electrical and Electronics Engineers* |
| LTL | *Linear Temporal Logic* |
| MAC | *Medium Access Control* |
| MC | *Model Checker* |
| MC | *Model Checking* |
| MF | Método Formal |
| MITL | *Metric Interval Temporal Logic* |
| PCTL | *Probabilistic Computation Tree Logic* |
| PLTL | *Probabilistic Linear Temporal Logic* |
| PRISM | *Probabilistic Symbolic Model Checker* |
| PROMELA | *Process Meta Language* |
| PTA | *Probabilistic Timed Automaton* |
| QoS | *Quality of Service* |
| RFC | *Request for Comments* |
| RTS | *Real Time System* |

RTS *Request to Send*

SDL *Specification and Description Language*

SIFS *Short Interframe Space*

SNR *Signal to Noise Ratio*

STA *Mobile Station*

TCTL *Timed Computation Tree Logic*

**SUMÁRIO**


**1 INTRODUÇÃO** .......... **11**
1.1 OBJETIVOS .......... 12
1.1.1 Objetivo Geral .......... 12
1.1.2 Objetivos Específicos .......... 12
1.2 JUSTIFICATIVA .......... 13
1.3 ESTRUTURA DO DOCUMENTO .......... 13
**2 *MODEL CHECKING*** .......... **15**
2.1 VISÃO GERAL SOBRE O *MODEL CHECKING* .......... 15
2.2 SISTEMAS DE TRANSIÇÃO .......... 17
2.3 PROPRIEDADES LINEARES .......... 20
2.3.1 Segurança .......... 20
2.3.2 Vivacidade .......... 20
2.3.3 *Fairness* .......... 21
2.4 LÓGICAS TEMPORAIS: LTL, CTL E TCTL .......... 21
2.4.1 Lógica Temporal Linear .......... 21
2.4.2 Lógica de Árvore de Computação .......... 23
2.4.3 Lógica de Árvore de Computação Temporal .......... 24
2.4.4 LTL vs CTL .......... 25
2.5 AUTÔMATO TEMPORAL .......... 27
**3 *MODEL CHECKERS*** .......... **29**
3.1 SPIN .......... 29
3.2 PRISM .......... 30
3.3 UPPAAL .......... 33
3.4 UPPAAL SMC .......... 38
3.5 DIVINE .......... 38
3.6 COMPARAÇÃO ENTRE *MODEL CHECKERS* .......... 39
**4 PROTOCOLOS MAC PARA REDES SEM FIO** .......... **40**
4.1 PROTOCOLO MAC PARA REDES *HALF DUPLEX* SEM FIO .......... 40
4.1.1 DCF .......... 41
4.1.2 *Backoff* .......... 41
4.1.3 Processo de Confirmação de Transmissão .......... 42
4.1.4 Mecanismo de Sensoriamento RTS/CTS .......... 43
4.2 PROTOCOLO MAC PARA REDES *FULL DUPLEX* SEM FIO .......... 44
**5 TRABALHOS RELACIONADOS** .......... **46**
5.1 MÉTODOS FORMAIS, PROTOCOLOS DE COMUNICAÇÃO E *NETWORKING* 46
5.2 *MODEL CHECKING* E PROTOCOLOS DE REDE SEM FIO .......... 48
5.2.1 *Model Checking* Probabilístico Aplicado em Redes de Sensores .......... 48
5.2.2 *Model Checking* Probabilístico e o Padrão 802.11 da IEEE .......... 49
5.2.3 *Model Checking* e o Protocolo AMCLM .......... 49
5.2.4 *Model Checking* e o Padrão 802.11 da IEEE, DCF .......... 50
5.2.5 *Model Checking* o Padrão 802.11 da IEEE, DCF e PCF .......... 51
5.3 VISÃO GERAL DOS TRABALHOS RELACIONADOS .......... 52
**6 DESENVOLVIMENTO** .......... **54**
6.1 PADRÕES PARA MODELAGEM .......... 54
6.2 SUPOSIÇÕES .......... 55
6.3 ESTRUTURA DO MODELO .......... 56

6.4 ENVIO DE MENSAGENS ... 57
6.5 *BACKOFF* ... 60
6.6 PROCESSO DE CONFIRMAÇÃO DE TRANSMISSÃO ... 61
6.7 TRATAMENTO DE COLISÃO ... 64
6.8 MODELO SIMPLIFICADO (SEM *PRIMARY TIMER* E *ACKTIMEOUT*) ... 67
6.9 APRIMORAMENTOS ... 69
6.9.1 Relógio não Reinicializado ... 69
6.9.2 Ausência de Prioridade ... 69
6.9.3 Números Aleatórios ... 70
6.9.4 Variável atribuída com valor *out of range* ... 71
6.9.5 Informações Técnicas ... 71
**7 RESULTADOS ... 73**
7.1 DETALHES DO AMBIENTE DE VERIFICAÇÃO ... 73
7.2 GERAÇÃO DOS INVERVALOS DE TEMPO ... 73
7.3 PARÂMETROS DE VERIFICAÇÃO DO UPPAAL ... 74
7.4 RESULTADOS ... 74
7.4.1 *Deadlock* ... 74
7.4.2 *Nenhuma transmissão em* $T_n$ ... 75
7.4.3 *k* transmissões em $T_n$ ... 76
7.4.4 *Transmissões simultâneas* ... 76
7.4.5 Transmissões simultâneas terminam ao mesmo tempo ... 77
7.4.6 ACK na transmissão primária ... 78
7.4.7 ACK na transmissão secundária ... 78
7.4.8 ACK simultâneo ... 79
7.5 VISÃO GERAL DOS RESULTADOS ... 79
**8 CONCLUSÃO ... 81**
**REFERÊNCIAS ... 83**
**APÊNDICE A – DOCUMENTAÇÃO DO MODELO ELABORADO ... 86**
**ANEXO A – MODEL CHECKING E O PADRÃO 802.11 DA IEEE, DCF ... 93**

# 1 INTRODUÇÃO

Programas de computador estão cada vez mais complexos e sendo utilizados em diferentes setores como em bolsas de ações e aviação (BAIER; KATOEN, 2008). Erros de especificação do funcionamento de um sistema, nestes e em outros cenários, podem causar perdas financeiras e, em cenários improváveis, mortes. Consequentemente, é crítico o desenvolvimento de tais sistemas, especialmente sistemas RTS (*Real Time Systems*) e concorrentes e distribuídos onde garantir ausência de erros é uma tarefa complexa. Análise estática de código e teste de unidades ajudam na realização dessa tarefa, mas são atividades complexas e trabalhosas de serem realizadas em sistemas concorrentes e distribuídos, uma vez que diferentes funções podem ser executadas simultaneamente tornando-se difícil entender como uma afeta a outra.

Métodos formais podem auxiliar na verificação e validação de programas e protocolos pelo uso de diferentes formalismos lógicos para provar a corretude de um programa. Alguns exemplos são especificação formal, provadores de teoremas, verificação de modelos (*model checking*). Destes formalismos, destaca-se o *model checking*, o qual possui vantagens como ampla aplicabilidade, geração de contraexemplos, sólida base lógica-matemática e curva de aprendizagem relativamente simples. Sua adoção é crescente em diversas áreas e setores tais como: redes (JENSEN; LARSEN; SKOU, 1996); verificação de hardware (BENTLEY, 2001); e veículos espaciais (BAIER; KATOEN, 2008).

Uma área onde métodos formais e *Model Checking* têm sido aplicados é a de protocolos de comunicação, sendo Bochmann (1978) o trabalho seminal da área, no qual é proposto o uso de modelos para especificação e validação de protocolos de comunicação. Desde então artigos foram publicados sobre a verificação de protocolos de comunicação (KWIATKOWSKA; NORMAN; SPROSTON, 2002); (FRUTH, 2006); (MATEO *et al.*, 2015). No entanto, protocolos de rede sem fio não têm sido tanto o foco de tais verificações principalmente porque requerem um alto custo inicial e geralmente são complexos (BABICH; DEOTTO, 2002; QADIR; HASAN, 2015). Recentemente, dois trabalhos relacionados a protocolos de acesso ao meio em redes sem fio foram publicados, Hammal *et al.* (2014) e Rosas (2014). O protocolo em questão é o DCF (*Distributed Coordination Function*), o qual especifica as etapas de acesso ao meio a serem seguidas pelos nós conectados à rede, implementado no padrão 802.11. A verificação formal desse protocolo pode ser vista em Rosas (2014) onde *model checking* e lógica temporal foram utilizados para modelar o protocolo verificando algumas propriedades, como ausência de *deadlock* e a possibilidade de uma estação jamais transmitir dados dentro de um intervalo de tempo $T_n$. Em Hammal *et al.* (2014) verificou-se formalmente uma versão modificada chamada *Adaptive Multi-Services Cross-Layer Mac Protocol* (AMCLM) que estabelece um limite *Signal to Noise Ratio* (SNR) para nós móveis. Nós que apresentam valores de transmissão abaixo deste limite são temporariamente desconectados da rede por apresentarem alta relação sinal-ruído na rede, o que acaba interferindo no desempenho da mesma. Outro trabalho encontrado foi o de Barboza *et al.* (2008) onde o

protocolo de controle de acesso ao meio 802.11 é formalmente verificado. O diferencial desse trabalho para o de Rosas (2014) é que o modo opcional denominado *Point Coordination Function* (PCF) é também verificado e o modo *Request to Send/Clear to Send* (RTS/CTS) é incluído para a *Distribution Coordination Function* (DCF). O *model checker* empregado foi o UPPAAL. Algumas propriedades verificadas incluem a ausência de *deadlock* e que a DCF não interfere no funcionamento da PCF.

Uma abordagem recente é o desenvolvimento de protocolos de rede *full duplex*. Protocolos desta natureza possibilitam o envio e o recebimento de dados ao mesmo tempo, o que pode melhorar significativamente a vazão de uma rede sem fio (JAIN *et al.*, 2011). Tanto o protocolo verificado por Hammal *et al.* (2014) quanto o por Rosas (2014) são versões *half duplex*, ou seja, somente uma transmissão ocorre no meio compartilhado (JAIN *et al.*, 2011). Assim, o presente trabalho objetiva estender o trabalho de Rosas (2014) tendo como objetivo a verificação formal através de *Model Checking* de uma outra variação do DCF padrão IEEE 802.11 usando o *model checker* UPPAAL (BEHRMANN; DAVID; LARSEN, 2006). O protocolo é o proposto em Jain *et al.* (2011), o qual permite que duas estações simultaneamente enviem dados no mesmo meio evitando o problema do terminal escondido através do envio de *busy tones* (definido no Capítulo 4). Apesar de haver outras propostas de protocolos *full duplex*, esse protocolo foi escolhido por possuir apresentar uma abordagem consensual na comunidade de rede que é a utilização de *busy tones* em tais transmissões.

## 1.1 OBJETIVOS

O objetivo geral e os objetivos específicos são descritos nas subseções a seguir.

### 1.1.1 Objetivo Geral

Realizar a verificação formal de um protocolo de controle de acesso ao meio *full duplex* usando *Model Checking*.

### 1.1.2 Objetivos Específicos

Os objetivos específicos são descritos a seguir:

1. Estudar os fundamentos lógicos de *Model Checking*;
2. Investigar alguns *model checkers*, programas que realizam verificação formal através de *model checking*, dentre eles o UPPAAL;

3. Estudar o protocolo *full duplex* (FD);
4. Modelar formalmente o protocolo FD em um *model checker*.
5. Especificar e verificar propriedades formais referentes ao protocolo.

## 1.2 JUSTIFICATIVA

Erros na especificação de programas e protocolos podem gerar implementações inconsistentes e não confiáveis. Cita-se um exemplo recente onde um avião 787 da Boeing apresentou um erro no qual o avião poderia perder toda a energia elétrica independentemente da situação (ADMINISTRATION, 2015). Isso ocorre caso o avião tenha todas as suas unidades de controle dos geradores de energia ligadas por mais de 248 dias consecutivos. Certificar, então, o funcionamento adequado de programas traz garantias para sua aplicabilidade como confiabilidade e segurança. Assim, o uso de métodos formais neste cenário poderia colaborar na detecção de erros e, eventualmente, nas suas correções.

Constata-se que a utilização de métodos formais é justificada pela capacidade de encontrar falhas comprometedoras em sistemas, sobretudo em sistemas críticos de tempo real, onde erros podem ser catastróficos. Apesar do presente trabalho não lidar com situações delicadas como a exposta acima, provar certas propriedades é desejável para garantir o correto funcionamento de um protocolo responsável por estabelecer a comunicação de dispositivos móveis, como é o caso de um protocolo de rede sem fio que poderia ser usado para comunicação em aviões, por exemplo.

Protocolos de rede, em razão de suas características, encaixam-se de modo adequado na verificação através de *model checking*. Algumas características como a presença de elementos distintos operando paralelamente, comunicação entre os mesmos, sincronia de operações, entre outras, são comumente encontradas e podem ser representadas nesse formalismo de maneira adequada. Consequentemente, é possível afirmar que a verificação formal de um protocolo de rede *full duplex* é pertinente dentro do contexto aqui descrito, principalmente pelo fato de que ainda não possui verificação formal na literatura.

## 1.3 ESTRUTURA DO DOCUMENTO

O processo de *model checking* é explicado no Capítulo 2 juntamente com as lógicas temporais comumente utilizadas para a formalização de propriedades. Alguns *model checkers* são apresentados e comparados no Capítulo 3. O Capítulo 4 explana o protocolo de controle de acesso ao meio em redes sem fio atualmente utilizado e uma versão *full duplex* (ainda não padronizada) do mesmo. O Capítulo 5 analisa trabalhos relacionados enquanto o Capítulo 6 relata

o desenvolvimento do trabalho explicando o modelo por partes elencando alguns aprimoramentos para o trabalho de Rosas (2014). Os resultados obtidos são apresentados no Capítulo 7. O Capítulo 8 conclui o presente trabalho sumarizando os resultados obtidos e indicando os trabalhos futuros. O Apêndice A documenta o modelo e o Anexo A contém o modelo utilizado como base para o desenvolvimento deste trabalho.

# 2 *MODEL CHECKING*

O presente capítulo trata do processo de *model checking* como um todo apresentando conceitos utilizados no processo. Um resumo deste capítulo pode ser visto a seguir. A Seção 2.1 explica o que é *model checking*. A Seção 2.3 define os tipos de propriedades que podem ser especificadas na verificação formal. Segue-se a Seção 2.2 que apresenta o conceito de sistema de transição, formalismo que pode ser usado para a modelagem de protocolos. A Seção 2.4 lida com as lógicas que são empregadas na especificação das propriedades apresentadas na Seção 2.3. A Seção 2.5 conclui o capítulo apresentando autômato temporal, formalismo utilizado na modelagem de protocolos no UPPAAL.

## 2.1 VISÃO GERAL SOBRE O *MODEL CHECKING*

Devido à complexidade sempre crescente na Tecnologia da Informação e Comunicação (*Information and communications technology* — ICT) sistemas precisam ser altamente confiáveis. Bolsas de valores e aviação são exemplos de áreas que dependem substancialmente de programas operando corretamente (BAIER; KATOEN, 2008). Erros nestes e em outros cenários podem custar recursos financeiros, ativos, e até vidas em uma situação caótica. Portanto, projetar e desenvolver sistemas de alta confiabilidade são atividades críticas, em especial para sistemas RTS. Além disso, testes de unidade e análise estática podem ser atividades demasiadamente difíceis de serem executadas em sistemas concorrentes, pois costumam ser complexos fazendo com que as atividades sejam altamente propensas a erro. *Model checking* auxilia neste processo laborioso garantindo que o sistema funcione de acordo com a especificação realizada.

*Model checking* é um método formal baseado em modelos capaz de verificar se um dado programa satisfaz ou não uma especificação definida em uma linguagem formal (BAIER; KATOEN, 2008). O programa (geralmente um protocolo) é representado por um modelo que consiste de estados (nós) e transições (ou arestas). Uma especificação é um conjunto de propriedades que o sistema deve possuir para operar corretamente. Exemplos de propriedades são a ausência de *deadlock* e vivacidade (*liveness*). A primeira significa que o programa nunca deve chegar em um determinado estado no qual não haja nenhuma alternativa, ou seja, onde nenhum progresso é possível. A segunda garante que o programa em algum momento realiza alguma operação para a qual foi designado. Tais propriedades são especificadas através de uma linguagem formal o que evita a introdução de ambiguidades na especificação. Um exemplo de linguagem formal comumente utilizada é a lógica temporal linear a qual possui os operadores da lógica booleana e adiciona dois operadores temporais, o que possibilita especificar a ordenação de eventos (detalhes no Subseção 2.4.1).

Ainda de acordo com Baier e Katoen o processo de verificação segue as seguintes etapas:

1. **Modelagem:** um modelo preciso é elaborado para retratar o sistema com estados identificando o comportamento e arestas a transição entre estados. Um conjunto de especificações é feito com relação ao comportamento esperado, e.g., um protocolo de comunicação deve responder uma mensagem em não mais que $x$ unidades de tempo;

2. **Execução:** executa-se um *model checker* para verificar se o modelo satisfaz as especificações;

3. **Análise:** se uma fórmula é satisfeita, passa-se para a verificação da próxima propriedade. Caso contrário, o modelo pode estar complexo demais para ser verificado (explosão de estados) ou a fórmula foi especificada incorretamente. Em ambos os casos um processo de revisão detalhado deve ser empregado para encontrar e reparar o erro.

A Figura 1 fornece uma visão geral do processo de *model checking*.

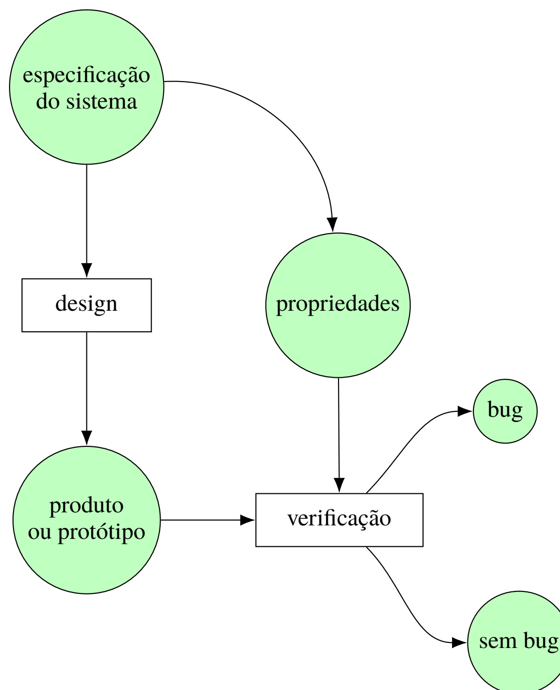



**Figura 1 – Ilustração do processo de *model checking***

**Fonte: Adaptado de Baier e Katoen (2008, p. 3)**

Baier e Katoen (2008, p. 14-16) elencam os prós e os contras do *model checking* os quais são elencados na Quadro 1.

<table>
<tr><th>Prós</th><th>Contras</th></tr>
<tr><td>Amplamente aplicável</td><td rowspan="2">Model checkers podem ter erros</td></tr>
<tr><td>Verificação parcial é posssível</td></tr>
<tr><td>Geração de contraexemplos</td><td rowspan="2">Problema da explosão de estados</td></tr>
<tr><td>Curva de aprendizagem suave</td></tr>
<tr><td>Crescente aplicação acadêmia e industrial</td><td rowspan="2">Verifica modelos somente</td></tr>
<tr><td>Sólida base lógica e matemática</td></tr>
</table>

**Quadro 1 – Prós e contras do *model checking***

**Fonte: Autoria própria**

O principal problema do *model checking* é a explosão de estados. A medida que um modelo torna-se complexo, a quantidade de estados a serem verificados cresce exponencialmente em relação ao número de estados e variáveis existentes no modelo pois, para cada estado, todos os valores possíveis de cada variável são verificados. Baier e Katoen (2008, p. 78) apontam que, para um sistema com dez estados, três variáveis booleanas, e cinco variáveis inteiras com valores dentro do intervalo $[0, 1 \ldots, 9]$, a quantidade de estados possíveis é de $10 \cdot 2^3 \cdot 10^5 = 8.000.000$, uma quantidade excessivamente alta para um sistema tão trivial.

## 2.2 SISTEMAS DE TRANSIÇÃO

Sistemas de transição são utilizados para realizar a modelagem de sistemas computacionais, especialmente sistemas concorrentes. Um sistema de transição é definido como uma tupla $(S, Act, \rightarrow, I, AP, L)$ (BAIER; KATOEN, 2008) onde:

- $S$ é um conjunto de estados;
- $Act$ é um conjunto de ações;
- $\rightarrow \subseteq S \times Act \times S$ é uma função que define a relação de transição entre estados;
- $I \subseteq S$ é o conjunto de estados iniciais;
- $AP$ é um conjunto de proposições atômicas que contém informações relevantes sobre os estados do sistema como, por exemplo, que um trem está passando por um cruzamento férreo;
- $L : S \rightarrow 2^{\text{AP}}$ é uma função que rotula elementos de $S$ que satisfazem as proposições atômicas $\in AP$.

Um exemplo simples é mostrado na Figura 2 onde três modelos representam componentes de um sistema de cruzamento ferroviário. O controlador gerencia a travessia de trens abaixando e levantando a cancela (BAIER; KATOEN, 2008). O sistema de transição representando o trem (Figura 2a) é definido por:

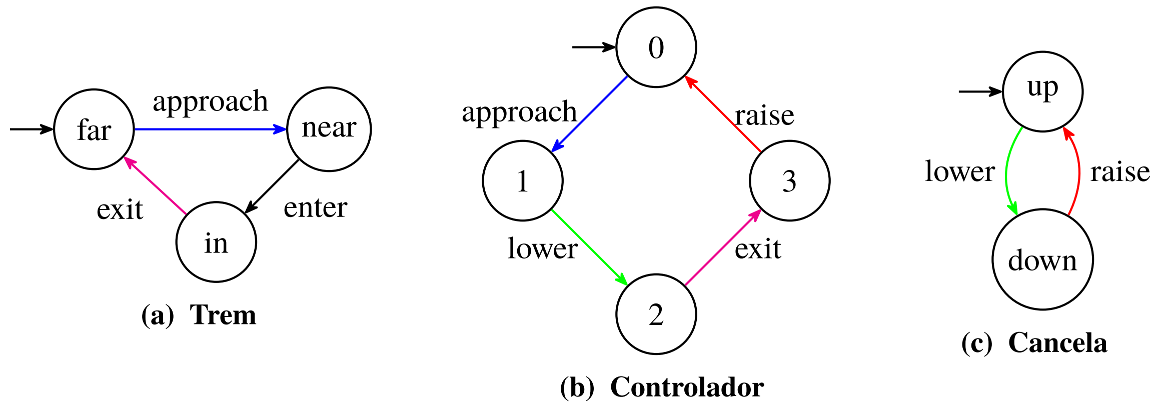



**Figura 2 – Sistemas de transição de um sistema de cruzamento ferroviário**

**Fonte: Adaptado de Baier e Katoen (2008, p. 53)**

- $S_{Train} = \{far, in, near\}$;
- $\rightarrow = (far \xrightarrow{approach} near), (near \xrightarrow{enter} in), (in \xrightarrow{exit} far)$;
- $Act_{Train} = \{approach, enter, exit\}$;
- $I_{Train} = \{far\}$;
- $AP_{Train} = \{crossing\}$;
- $L_{Train}(far) = L_{Train}(near) = \emptyset, L_{Train}(in) = \{crossing\}$.

A definição do sistema de transição representando o trem (Figura 2a) tem três estados possíveis: *far* representa um trem que está longe; *near* modela um trem aproximando-se da cancela; e *in* representa um trem atravessando a via. Quando se aproxima, o trem sinaliza o controlador (Figura 2b) para que a cancela seja abaixada (Figura 2c). Note que ambas as ações ocorrem de modo síncrono. Aspectos temporais são ignorados neste exemplo. Um exemplo de execução com todos os componentes da Figura 2 pode ser visto na Figura 3.

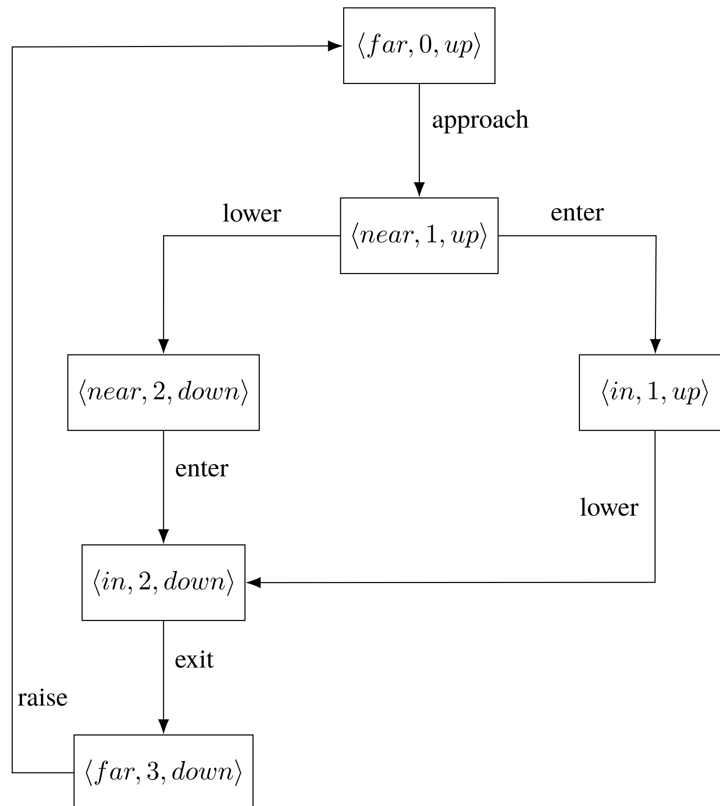



**Figura 3 – Sistema de transição parcial de um sistema de cruzamento ferroviário**

**Fonte: Adaptado de Baier e Katoen (2008, p. 54)**

Um *program graph* é similar a um sistema de transição mas pode conter transições condicionais. É definido como uma tupla $(Loc, Act, Effect, \to, Loc_0, g_0)$ onde:

- $Loc$ é um conjunto de estados;
- $Act$ é um conjunto de ações;
- $Effect : Act \times Eval(Var) \to Eval(Var)$, onde $Var$ é um conjunto finito de variáveis utilizadas nas transições condicionais;
- $\to \subseteq Loc \times Cond(Var) \times Act \times Loc$ define uma relação de transição entre os estados;
- $Loc_0$ é o conjunto contendo todos os estados iniciais;
- $g_0$ é o conjunto de condições iniciais.

No entanto, sistemas de transição não são capazes de representar aspectos temporais. Considerando o exemplo do cruzamento férreo visto anteriormente, por meio de um sistema de transição não é possível determinar que o trem levará cinco unidades de tempo para realizar o cruzamento e assim definir o tempo que a cancela deve permanecer fechada. Um determinado tipo de autômato (autômato temporal) (detalhes na Seção 2.5) introduz o conceito de relógios que são variáveis reais que medem o passar do tempo, essencial para RTSs (ALUR; DILL, 1994).

## 2.3 PROPRIEDADES LINEARES

Uma visão geral sobre propriedades lineares é dada nas próximas subseções. Na Subseção 2.3.1 é apresentado o conceito de segurança na especificação de propriedades enquanto a Subseção 2.3.2 apresenta propriedades de *liveness*. A Subseção 2.3.3 conclui a presente seção mostrando o conceito de *fairness* na formalização de propriedades.

### 2.3.1 Segurança

Quando um programa é desenvolvido, espera-se que este opere corretamente, o que engloba que durante a execução do programa nada de errado ocorra. Propriedades lineares de segurança são utilizadas para garantir que algo de errado nunca irá acontecer. Para isso, durante a verificação do modelo todos os estados possíveis são analisados para assegurar que, durante a execução, não há nenhum caminho onde a propriedade de segurança seja violada (BAIER; KATOEN, 2008). Um exemplo de propriedade pode ser vista em 2.1:

$$\forall i, j \geq 0 \; |\{ \; 0 \leq \; j \leq i \mid pay \in A_j\}| \geq |\{ \; 0 \leq j \leq i \mid drink \in A_j\}| \tag{2.1}$$

Esse exemplo, adaptado de Baier e Katoen (2008), ilustra uma máquina de dispensa de bebidas. Uma bebida deve ser dispensada ($drink$) somente após uma moeda ter sido inserida pelo usuário ($pay$). $A_j$ indica um estado qualquer do sistema.

### 2.3.2 Vivacidade

Outra propriedade esperada é que o sistema, de fato, realize alguma coisa, execute uma dada ação. Esse tipo de propriedade é referida como vivacidade (*liveness*) e afirma que, em algum momento, o sistema irá realizar algo *bom* (BAIER; KATOEN, 2008). Um exemplo dado é o caso de dois processos que precisam entrar em suas regiões críticas mutuamente exclusivas. Uma propriedade básica (2.2) é o fato de que, pelo menos uma vez durante a execução, ambos entrem em suas regiões críticas ($crit_1, crit_2$). Existe um conjunto AP de propriedades atômicas onde $AP = \{wait_1, \; crit_1, \; wait_2, \; crit_2\}$ e $A_j \subseteq AP$. Por sua vez, $crit_1$ e $crit_2$ são rótulos para representar propriedades atômicas que identificam estados de procesos que ingressam em regiões críticas onde $j$ e $k$ são índices que representam estados dos modelos indo de zero até infinito.

$$(\exists j \geq 0. \; crit_1 \in A_j) \wedge (\exists j \geq 0. \; crit_2 \in A_j) \tag{2.2}$$

Entretanto, tal propriedade não garante que, após entrar uma vez em sua região, o processo venha a entrar novamente. Refinando a propriedade a fim de expressar que sempre ambos entrem em suas respectivas áreas críticas resulta em (2.3). Ambas as propriedades podem ser encontradas em Baier e Katoen (2008, p. 122).

$$(\forall k \geq 0.\ \exists j \geq k.\ crit_1 \in A_j) \wedge (\forall k \geq 0.\ \exists j \geq k.\ crit_2 \in A_j) \tag{2.3}$$

### 2.3.3 *Fairness*

O último tipo de propriedade brevemente apresentada neste trabalho é *fairness* e está relacionada com a equidade do comportamento do sistema. Isso significa que o sistema deve ser *justo* com relação aos processos que participam dele e elimina certos comportamentos que são produzidos quando *interleaving* – escolha não determinística das possíveis ações dos sistemas executando em paralelo – é realizado no sistema, isto é, às vezes um processo pode ter sempre a prioridade sobre outro (BAIER; KATOEN, 2008).

## 2.4 LÓGICAS TEMPORAIS: LTL, CTL E TCTL

As seções seguintes explicam, brevemente, as lógicas temporais utilizadas no processo de especificação de propriedades. Lamport (1983, p. 2) salienta que o termo *temporal* não está relacionado com a capacidade de adicionar a medição de tempo, mas sim com a possibilidade de estabelecer sucessão temporal na ordenação dos eventos. A Subseção 2.4.1 introduz a lógica temporal linear (*Linear Temporal Logic*, LTL), que é utilizada para a especificação de propriedades temporais. A Subseção 2.4.2 apresenta lógica de árvore de computação (*Computation Tree Logic*, CTL), a qual especifica propriedades através de caminhos e estados. Por fim, a seção é concluída após a Subseção 2.4.3 que sucintamente apresenta uma variação da CTL com suporte a fórmulas temporais[1], a lógica de árvore de computação temporal (*Timed Computation Tree Logic*, TCTL).

### 2.4.1 Lógica Temporal Linear

Para a especificação de propriedades lineares necessita-se de uma linguagem formal capaz de expressá-las. Uma das primeiras lógicas criadas com tal fim é a lógica temporal linear (*Linear Temporal Logic*, LTL) que possui os operadores básicos da lógica clássica proposicional ($\neg$, $\wedge$) e adiciona dois operadores temporais: o operador *próximo* (*next*, $\bigcirc$) e o *até* (*until*, $\cup$). A

[1] No sentido de progressão temporal e não somente na ordenação dos eventos.

gramática da LTL é vista a seguir (2.4):

$$\varphi ::= true \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \cup \varphi_2 \tag{2.4}$$

O operador $\bigcirc$ é unário e indica que no próximo momento a fórmula $\varphi$ será verdadeira. $\cup$ é um operador binário e especifica que, em todos os momentos antes que $\varphi_2$ torne-se verdadeiro, $\varphi_1$ é sempre verdadeiro. $a$ é um elemento do conjunto de proposições atômicas $AP$. Dois outros operadores unários são derivados do operador $\cup$: o operador *agora ou em algum momento* no futuro ($\Diamond$) e o *agora e sempre* no futuro ($\Box$).[2] O primeiro operador é derivado da seguinte forma: $\Diamond\varphi \equiv true \cup \varphi$. O segundo operador é dado por $\Box\varphi \equiv \neg\Diamond\neg\varphi$. A Figura 4 ilustra, de modo intuitivo, a semântica dos operadores da LTL.

Um exemplo de fórmula LTL é $\Box(\neg crit_1 \vee \neg crit_2)$, a qual especifica que dois processos disputando acesso às suas regiões críticas **mutuamente exclusivas** jamais podem ocupá-las simultaneamente, em outras palavras, sempre somente um processo estará em sua região crítica. Uma outra fórmula LTL é $(\Box\Diamond wait_1 \rightarrow \Box\Diamond crit_1) \wedge (\Box\Diamond wait_2 \rightarrow \Box\Diamond crit_2)$, a qual especifica que, se infinitamente frequente um processo está esperando por sua vez para entrar na região crítica, infinitamente frequente irá entrar na mesma.

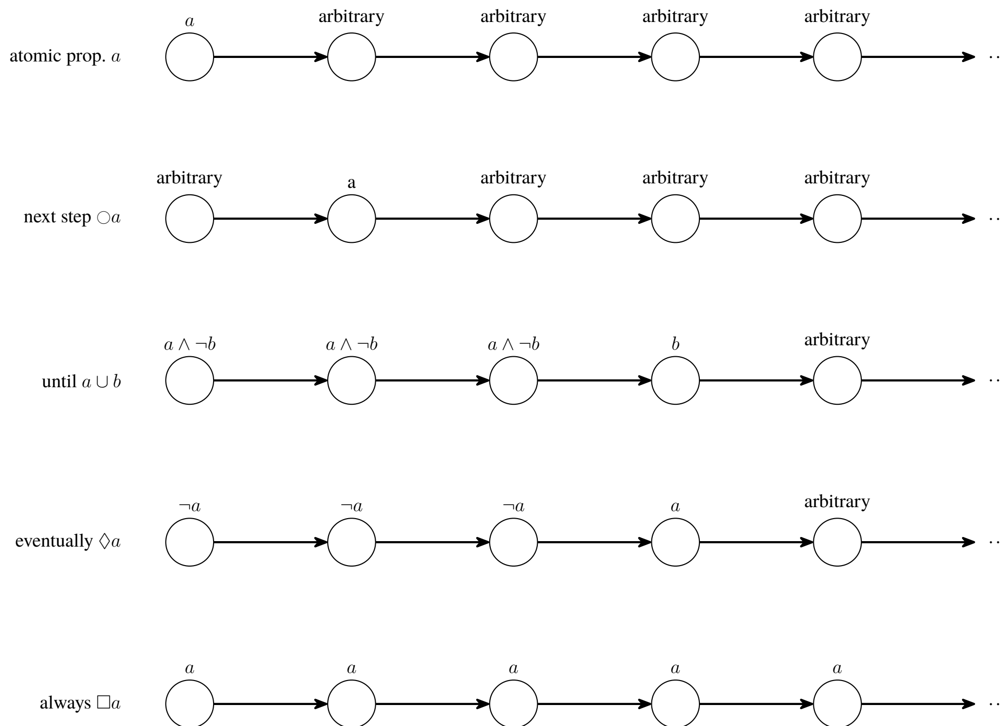



**Figura 4 – Representação visual dos operadores da LTL**

**Fonte: Adaptado de Baier e Katoen (2008, p. 233)**

[2] Os operadores $\Box$ e $\Diamond$ são oriundos da lógica modal e são conhecidos também como operadores: quadrado e diamante. Ademais, podem assumir diferentes semânticas dependendo do tipo de lógica em que são usados, *e.g.* lógica modal, deôntica, temporal, etc. Mas, em linhas gerais, o operador quadrado representa algo que é necessário, contrastantando com o diamante que representar que algo é possível.

### 2.4.2 Lógica de Árvore de Computação

A LTL é, de certa forma, limitada devido ao fato de que sempre há somente um sucessor temporal. A lógica de árvore de computação (*Computation Tree Logic*, CTL) é uma lógica temporal com ramificações que permite especificar propriedades relacionadas a vários caminhos possíveis a partir de um estado $s$.

A sintaxe da CTL para a especificação de propriedades é dividida em duas partes: uma para estados e outra para caminhos. A gramática de estados é definida por (2.5):

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi \tag{2.5}$$

onde $\exists$ e $\forall$ são quantificadores de caminhos, existencial e universal, respectivamente; $\varphi$ é uma fórmula de caminho e $a \in AP$ (conjunto de proposições atômicas). A CTL também possui o operador $\bigcirc$. Desse modo é possível derivar os operadores temporais presentes na LTL – $\Box$, $\Diamond$, $\cup$ – com exatamente os mesmos significados.

Para a especificação de fórmulas de caminhos a gramática é a seguinte (2.6):

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \cup \Phi_2 \tag{2.6}$$

onde $\Phi$, $\Phi_1$, e $\Phi_2$ são fórmulas de estados (definidas através da primeira gramática – 2.5). Um exemplo de fórmula CTL é $\forall\Box(\neg crit_1 \vee \neg crit_2)$ que especifica que, para todas as computações possíveis, somente um estado está na sua região crítica.

A Figura 5 ilustra o significado das fórmulas em CTL. A Figura 5a indica que existe ($\exists$) uma ramificação onde *black* é em algum momento ($\Diamond$) verdadeira. A Figura 5b especifica que há ($\exists$) uma ramificação na qual a proposição *black* é sempre ($\Box$) verdadeira. A Figura 5c representa que para todos os caminhos ($\forall$) *black* é em algum momento verdadeira ($\Diamond black$). A Figura 5d define que para todas as ramificações possíveis ($\forall$) *black* é sempre ($\Box$) satisfeita. A Figura 5e mostra que há caminhos ($\exists$) nos quais os estados são *gray* até ($\cup$) *black*. Por fim a Figura 5f define que para todos os caminhos possíveis ($\forall$) os estados são *gray* até que ($\cup$) a propriedade *black* seja satisfeita.

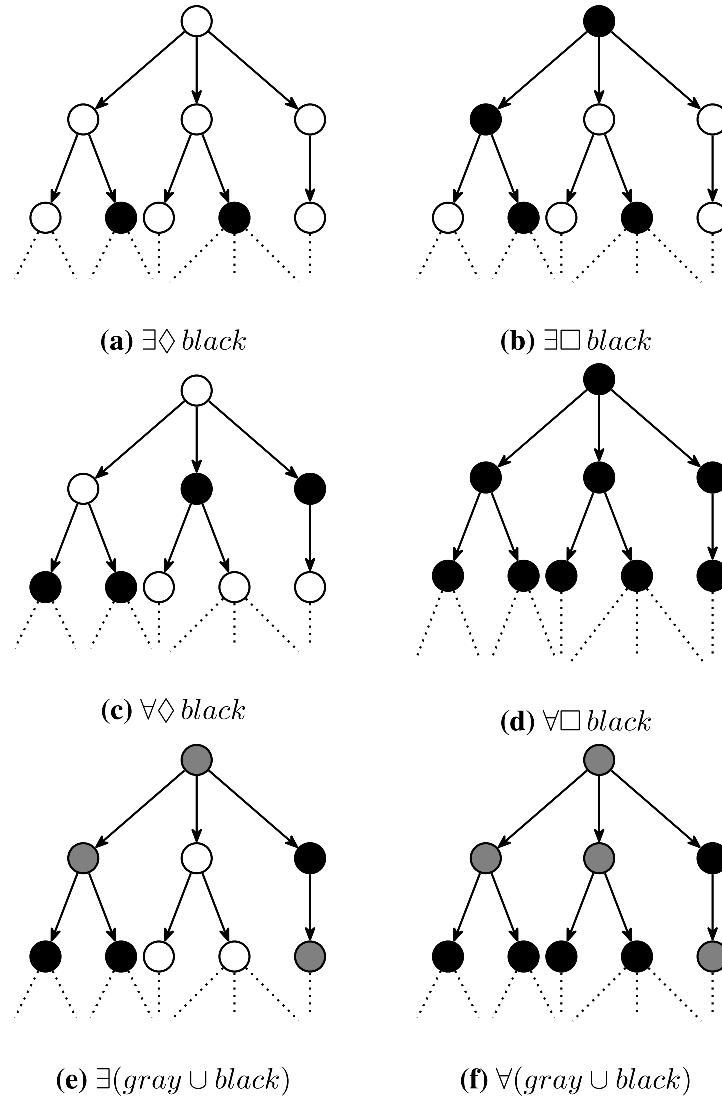



**Figura 5 – Representação visual dos operadores da CTL**

**Fonte: Adaptado de Baier e Katoen (2008, p. 322)**

### 2.4.3 Lógica de Árvore de Computação Temporal

Para a especificação de propriedades que necessitam da especificação de limites temporais a lógica de árvore de computação temporal[3] pode ser usada. A TCTL estende a CTL com a inclusão da possibilidade de uma propriedade ser satisfeita dentro de um intervalo de tempo.

A sintaxe da TCTL para a especificação de propriedades também é dividida em duas partes: uma para estados e outra para caminhos. A gramática para a primeira é definida por (2.7):

$$\Phi ::= true \mid a \mid g \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi \tag{2.7}$$

onde $a \in AP$ (conjunto de proposições atômicas), $g \in ACC(C)$ (conjunto de restrições sobre o conjunto de relógios $C$) e $\varphi$ é uma fórmula de caminho. Fórmulas de caminhos são especificadas

[3] No sentido também de medir o tempo.

pela seguinte gramática (2.8):

$$\varphi ::= \Phi_1 \cup^J \Phi_2 \tag{2.8}$$

onde $J \subseteq \mathbb{R}_{\geq 0}$. Como o operador *até* ($\cup$) agora é temporal no sentido de medição de tempo, os operadores dele derivados ($\Diamond$ e $\Box$) também o passam a ser.

Um exemplo de fórmula TCTL é $\forall\Box((on \wedge x = 0) \rightarrow (\forall\Box^{\leq 1} on \wedge \forall\Diamond^{>1} off))$ reproduzida de Baier e Katoen (2008, p. 700). Esta fórmula especifica que, para todos os caminhos possíveis e sempre nestes ($\forall\Box$), um interruptor, uma vez ligado ($on \wedge x = 0$), irá permanecer ligado por, pelo menos uma unidade de tempo ($\forall\Box^{\leq 1} on$) antes de desligar ($\forall\Diamond^{>1} off$).

### 2.4.4 LTL vs CTL

Apesar de LTL e CTL serem usadas para a especificação de propriedades, elas não possuem a mesma expressividade. Há propriedades que podem ser expressas em LTL e não em CTL e vice versa. Há também propriedades que podem ser especificadas pelas duas no entanto.

Antes de comparar a expressividade de ambas, é preciso entender o que significa que um dado estado satisfaz uma fórmula LTL. Como a LTL considera as propriedades sobre caminhos, uma dada fórmula $\phi$ expressa em LTL é satisfeita por um estado $s$ quando, a partir de $s$, todos os caminhos possíveis satisfazem $\phi$. Percebe-se então que a semântica para fórmulas LTL é similar ao operador $\forall$ da CTL.

Primeiro uma fórmula expressível em CTL e não em LTL. Na Figura 6 tem-se um simples sistema de transição. Para $s_0$ a seguinte fórmula CTL é satisfeita: $\exists\Diamond a$, isto é, existe um caminho onde em algum momento a proposição atômica $a$ é satisfeita. Como a LTL não possui operadores relacionados a caminhos, a fórmula pode ser escrita como $\Diamond a$, também considerando $s_0$. Contudo, essa fórmula não é verdadeira para $s_0$, isso porque, apesar de a sequência de estados $(s_0, s_1)$ satisfazê-la, a sequência $(s_0, s_2)$ não satisfaz: em $s_2$ é verdade que $\neg a$.

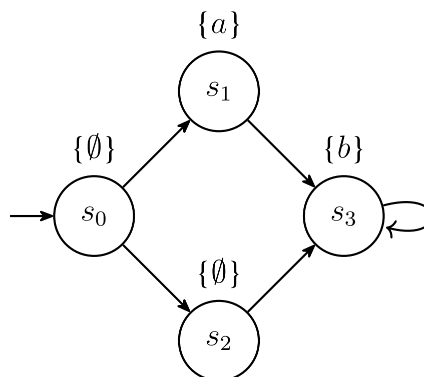



**Figura 6 – Sistema de transição para uma fórmula CTL não expressível em LTL**

**Fonte: Autoria própria**

Em seguida uma fórmula que pode ser expressa em LTL e não em CTL. Considere a fórmula $\phi$ como sendo $\Diamond(a \wedge \bigcirc a)$ interpretada sobre o sistema de transição da Figura 7. $\phi$ é claramente satisfeita em $s_0$, pois tanto os caminhos (sequência de estados) $(s_0, s_2, s_3)$ e $(s_0, s_1)$ satisfazem $\phi$. Uma representação dessa fórmula em CTL é $\forall\Diamond(a \wedge \forall \bigcirc a)$. No entanto, a equivalência não é preservada uma vez que o caminho $(s_0, s_2, (s_2^{\omega}))$ viola a fórmula, onde $\omega$ representa uma ou mais iterações.

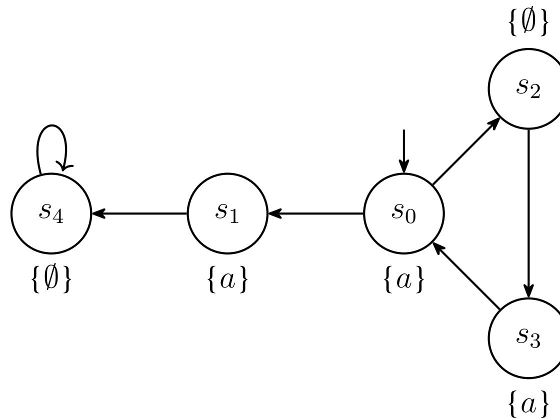



**Figura 7 – Sistema de transição para uma fórmula LTL não expressível em CTL**

**Fonte: Adaptado de Baier e Katoen (2008)**

A Figura 8 mostra um sistema de transição sobre o qual fórmulas equivalentes LTL e CTL são interpretadas. Agora tanto a fórmula $\Box a$ como $\forall\Box a$ são satisfeitas em $s_0$.

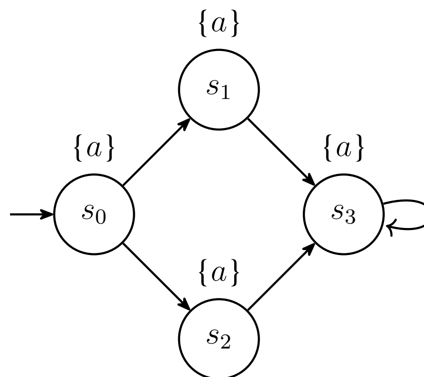



**Figura 8 – Sistema de transição para uma fórmula expressível em LTL e CTL**

**Fonte: Autoria própria**

## 2.5 AUTÔMATO TEMPORAL

Protocolos de rede exibem características temporais, por exemplo, se uma dada estação não responder dentro de $x$ unidades de tempo assume-se que houve erro na comunicação e o processo de comunicação precisa ser reiniciado (IEEE Computer Society; LAN/MAN Standards Committee; Institute of Electrical and Electronics Engineers, 2007). Embora sistemas de transição sejam um formalismo expressivo, eles não têm a capacidade de medir a progressão do tempo nos sistemas modelados. Assim, acabam não sendo adequados para a modelagem de RTSs. Para superar tal dificuldade, Alur e Dill (1994) estenderam o conceito de autômatos introduzindo relógios. Um relógio nada mais é que uma variável real capaz de medir o passar do tempo em um sistema. Um autômato temporal (*timed automaton*, TA) é definido como um sistema de transição que possui restrições temporais sendo uma tupla $(L, L^0, \Sigma, X, I, E)$ onde:

- $L$ é um conjunto finito de *locations*[4];
- $L^0$ é um conjunto finito contendo as *locations* iniciais ($L^0 \subseteq L$);
- $\Sigma$ é um conjunto de rótulos de transições[5];
- $X$ é um conjunto finito de relógios $x$;
- $I$ atribui uma restrição do conjunto de restrições de relógios $\Phi(X)$ a uma *location*. $\varphi \in \Phi$ possui a seguinte gramática: $\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2$ onde $x \in X$ e $c$ é uma constante $\in \mathbb{Q}$;
- $E \subseteq L \times \Sigma \times 2^{x} \times \Phi(X) \times L$ é um conjunto de transições. Uma transição $(s, a, \varphi, \lambda, s')$ representa uma transição de $s$ para $s'$ sobre a ação $a$ restringida pela restrição de relógio $\varphi$ reiniciando um conjunto de relógios $\lambda \subseteq X$. Uma transição pode representar a passagem de tempo, $(s, v) \xrightarrow{\delta} (s, v + \delta)$ ou uma mudança de *location*, na qual $(s, v) \xrightarrow{a} (s', v[\lambda := 0])$, o que representa a mudança de um estado $s$ para $s'$ com os relógios pertencentes a $\lambda$ sendo reiniciados com o valor 0.

Complementarmente, pode-se ter um conjunto de proposições atômicas ($AP$) e uma função de rotulação $L : L \rightarrow 2^{AP}$ que atribui a *locations* elementos do conjunto $AP$ (BAIER; KATOEN, 2008). Função de rotulação é utilizada para facilitar a interpretação das possíveis *locations* de um sistema, o que acaba também auxiliando no processo de *model checking*.

Um autômato temporal simples pode ser visto na Figura 9 (ALUR, 1999). O autômato tem quatro *locations*, $s_0, s_1, s_2, s_3$, sendo $s_0$ o *location* inicial e dois relógios, *x* e *y*. A reinicialização de relógios pode ser vista na transição de $s_0$ para $s_1$ onde o relógio *x* é atribuído com

[4] Um estado é a *location* atual mais o valor atual do relógio. Ver <http://cs.stackexchange.com/questions/48811/why-are-states-called-locations-in-timed-automata/49005>

[5] Possibilita o emprego de transições rotuladas, o que auxilia na modelagem de sistemas.

valor zero. Na *location* $s_2$ o tempo passado não pode ser maior que uma unidade (restrição *x < 1*). A *location* $s_0$ só pode ser atingida após o tempo decorrer mais que duas unidades de tempo (transição *d, y > 2*). *c* identifica uma transição rotulada.

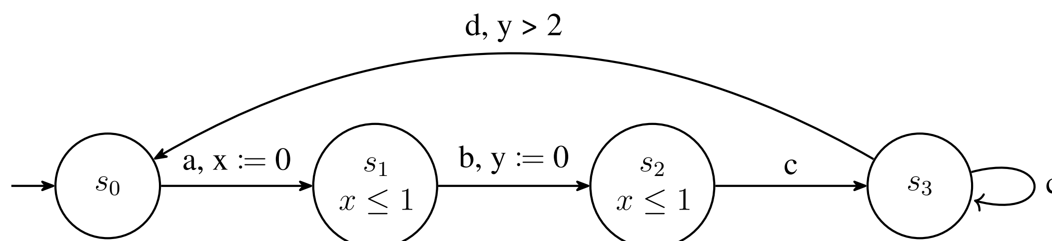



**Figura 9 – Autômato temporal simples**

**Fonte: Adaptado de Alur (1999, p. 9)**

# 3 *MODEL CHECKERS*

Neste capítulo são apresentados alguns programas capazes de verificar se um modelo satisfaz a um conjunto de propriedades especificadas por meio de lógica temporal, os chamados *model checkers*. Tais programas possuem diversas aplicações e são reconhecidos na comunidade científica. O capítulo inicia-se com a Seção 3.1 que introduz o *model checker* SPIN, um dos pioneiros na área de verificação formal. Segue-se a Seção 3.2 que, elaborada tendo como referência o trabalho de Kwiatkowska, Norman e Parker (2011) e Oxford (2015), explica o PRISM e alguns de seus conceitos . A Seção 3.3 descreve o UPPAAL. A Seção 3.4 apresenta sucintamente uma extensão do UPPAAL chamada SMC. A Seção 3.5 de modo breve expõe um *model checker multicore* chamado *DIVINE*. Na última parte, a Seção 3.6 conclui o presente capítulo com uma comparação entre os *model checkers* aqui destacados.

## 3.1 SPIN

SPIN é um *model checker* distribuído de forma *open source* para a verificação de sistemas, executável a partir de um terminal ou de uma GUI (*Graphical User Interface*) chamada de iSpin. Têm sido aplicado em diversas áreas de estudos teóricos e práticos, sendo dois exemplos proeminentes a verificação de um controlador de um veículo espacial (HAVELUND; LOWRY; PENIX, 2001) e de um sistema distribuído *multi-threaded* (STOLLER, 2000). Outra aplicação mais recente é a sua utilização na verificação formal de diagramas de sequência da UML 2.0 (LIMA *et al.*, 2009).

Basicamente o SPIN pode ser usado de duas formas (HOLZMANN, 1997): simulação interativa e verificação formal. Na simulação interativa o modelo é executado. Na verificação formal é gerado um programa (linguagem C) capaz de verificar se o modelo satisfaz as especificações informadas.

A modelagem é feita usando uma linguagem textual chamada PROMELA (*Process Meta Language*) a qual modela o comportamento do sistema. Propriedades referentes ao modelo são especificadas em LTL.

A estrutura de funcionamento é descrita na Figura 10. O analisador da linguagem PROMELA é utilizado nas versões *cli* (*command line interface*) e gráfica. Comunica também as fórmulas LTL (representadas por autômatos de Büchi)[1] especificadas no modelo para o módulo responsável, *LTL Parser and Translator*. Uma análise sintática é feita e erros serão reportados caso encontrados. Se não houver, passa-se para a próxima etapa que pode ser uma simulação interativa ou a geração de um programa (linguagem C) capaz de verificar formalmente o modelo

[1] Um tipo de autômato $\omega$ que aceita uma sequência infinita de entrada se, pelo menos uma vez, um estado terminal for visitado.

(parâmetro escolhido pelo usuário).

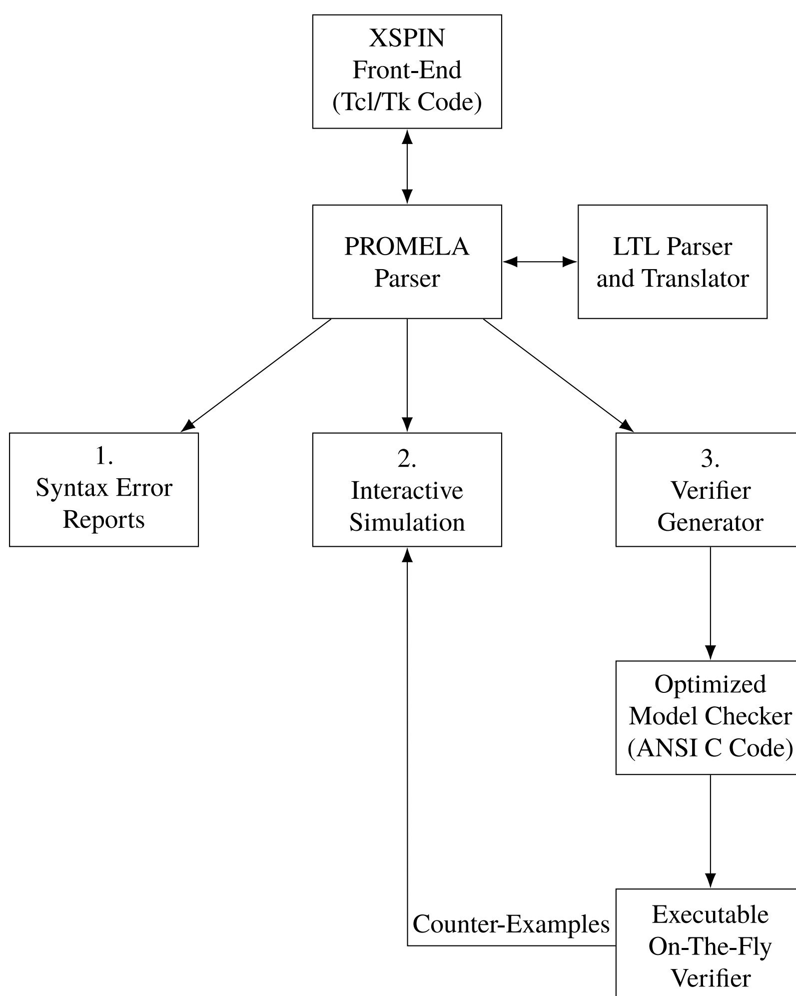



**Figura 10 – Estrutura interna do SPIN**

**Fonte: Adaptado de Holzmann (1997, p. 2)**

## 3.2 PRISM

PRISM (*Probabilistic Symbolic Model Checker*) é um *model checker* probabilístico multiplataforma desenvolvido em Java e C++ na Universidade de Oxford largamente aplicado em diversas áreas como biologia, protocolos de comunicação, algoritmos quânticos entre outras (KWIATKOWSKA; NORMAN; PARKER, 2011). É executado a partir da linha de comando ou através de uma GUI e aceita como modelos cadeias de Markov (com algumas variantes), processos decisórios de Markov e autômatos temporais probabilísticos.

Há duas linguagens no PRISM, uma para modelagem outra para especificação. Modelos são especificados na clara e intuitiva linguagem textual PRISM que possui um conjunto de identificadores, permite a declaração de constantes e tem os demais requisitos necessários para a

modelagem dos formalismos suportados. Especificações são realizadas na linguagem (também textual) de especificação de propriedades PRISM que subsume lógicas como LTL, CTL, PCTL.

Um modelo consiste de um ou mais módulos e possui um tipo específico, e.g., Autômato Temporal Probabilístico[2] (PTA), cadeia de Markov. Cada módulo possui declaração de variáveis locais, inteiras ou booleanas, seguida por uma série de comandos que descrevem ações possíveis caso as condições para suas respectivas execuções sejam satisfeitas. As variáveis também são utilizadas para representar os estados do sistema sendo sempre públicas (mesmo definidas localmente) porém só escritas pelo módulo que as possui. Variáveis globais também estão disponíveis caso seja preciso ter uma variável onde todos os módulos possam escrever.

A Figura 11a mostra um simples modelo que possui: a definição do tipo (PTA), a criação de um módulo *M*, as variáveis locais inteiras *s* e *x*, a atribuição de variantes aos primeiro e último estados, e as transições probabilísticas entre os estados. A Figura 11b representa visualmente o mesmo. A Figura 12 ilustra a organização geral de uma verificação formal no PRISM.

As propriedades podem ser analisadas de modo assertivo ou quantitativo. Um operador introduzido pela linguagem, o operador probabilístico *P*, calcula a probabilidade de uma dada propriedade ser verdadeira e pode ser usado nestes dois modos. Exemplos de propriedades são $P \geq 1$ [ $F$ “terminate” ] e $P_{max} =?$ [ $F \leq T\ messages_lost > 10$ ] onde a primeira especifica se é possível que o sistema termine sua execução com probabilidade um e a segunda calcula a probabilidade máxima de mais de dez mensagens serem perdidas até o tempo $T$. $F$ representa a probabilidade de uma propriedade ser verdadeira a partir do estado inicial.

[2] Autômato temporal com a adição de probabilidades sobre as suas transições. As probabilidades referem-se tanto à possibilidade de uma transição de passagem de tempo quanto às transições possíveis para outros estados a partir de um certo estado $s$.

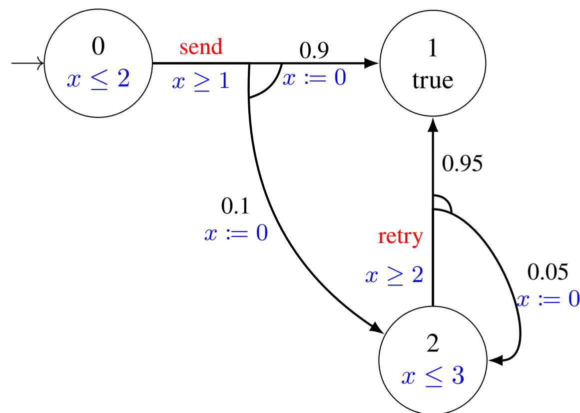
```
pta # probabilistic timed automata ('#' marks a comment)

module M
    s : [0..2] init 0; # s represents automaton states
    x : clock; # used in invariant states to measure elapsed time

    invariant # states invariants
        (s=0 => x<=2) &
        (s=2 => x<=3)
    endinvariant

    # named transition 'send' and 'retry'
    [send] s=0 & x>=1 -> 0.9:(s'=1)&(x'=0) + 0.1:(s'=2)&(x'=0);
    [retry] s=2 & x>=2 -> 0.95:(s'=1) + 0.05:(s'=2)&(x'=0);
endmodule
```

**(a) Modelo na linguagem PRISM**




**(b) Modelo representado em PTA**

**Figura 11 – Linguagem PRISM e PTA**

**Fonte: Adaptado de Oxford (2015)**

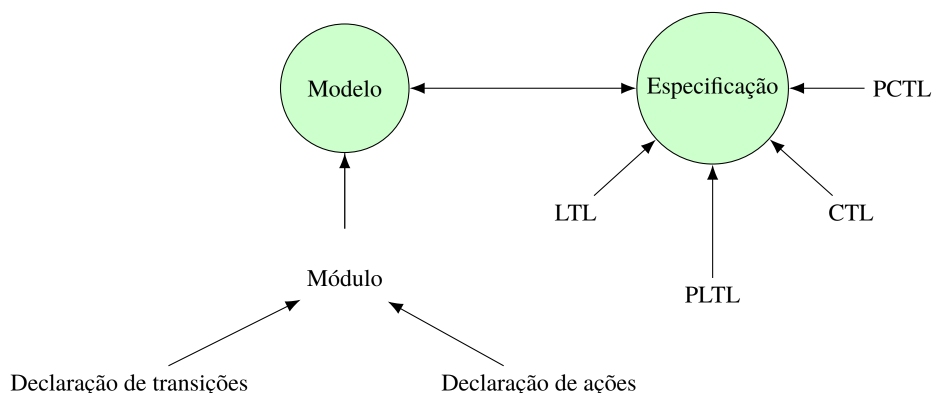



**Figura 12 – Visão geral do *model checking* no PRISM**

**Fonte: Autoria Própria**

## 3.3 UPPAAL

UPPAAL é um *model checker* capaz de verificar sistemas de tempo real e protocolos de comunicação desenvolvido pelo Departamento de Tecnologia da Informação da Universidade Uppsala na Suécia e o Departamento de Ciência da Computação da Universidade de Aalborg na Dinamarca. A versão atualmente disponível para download é a 4.1.19 de 2014 (*snapshot*).

Algumas de suas funcionalidades incluem o suporte a transições condicionais, invariantes em *locations* (utilizadas para medir o tempo passado enquanto em uma certa *location*) e comunicações síncronas por meio de canais. Outras características incluem a declaração de constantes, variáveis (locais ou globais), *broadcast*, vetores e funções, e simulação passo-a-passo (BEHRMANN; DAVID; LARSEN, 2006), (VAANDRAGER, 2011). Autômatos temporais são utilizados para a modelagem. Para a especificação de propriedades é usada uma versão simplificada da TCTL que, basicamente, engloba quatro tipos de propriedades lineares:

- **atingibilidade** (*reachability*) ($E <> \phi$)[3]: *E* um caminho no qual, em algum momento **<>**, há um estado onde $\varphi$ é satisfeita;
- **segurança** ($A[]\varphi$ ou $E[]\varphi$): a primeira verifica que em todos os caminhos possíveis *A*, em todos os estados ($[]$)[4] $\varphi$ é satisfeita. A segunda verifica que existe um caminho possível (*E*), onde em todos os estados ($[]$) uma fórmula $\varphi$ é satisfeita.
- ***liveness*** ($A <> \phi$): para todos os possíveis caminhos $A$, em algum momento $<>$ $\varphi$ é satisfeita.
- ***leads to*** ($\phi \leadsto \psi$): sempre que $\phi$ é satisfeita, em algum momento $\psi$ também será.

Nas três imagens que seguem são apresentadas especificações de um exemplo detalhado na sequência desta seção. A Figura 13 exibe a janela de edição do UPPAAL, onde é possível construir um modelo usando autômatos temporais, especificar as declarações globais e locais. A Figura 14 mostra a aba do verificador com a especificação de algumas propriedades. É nesta aba que ocorre a formalização e verificação das propriedades. O círculo é colorido com vermelho caso a a propriedade não seja satisfeita e com verde caso seja. A aba de simulação de modelos é vista na Figura 15. Nesta aba também é possível analisar os contraexemplos gerados pela ferramenta durante a verificação. A Figura 16 mostra a definição (global) de variáveis canais (para comunicação entre modelos) e constantes. Além da interface gráfica, há uma versão *cli* chamada *verifyta* adequada para a verificação em servidores.

---

[3] No UPPAAL não há espaço entre $E$ ou $A$ e $<>$ ou $[]$.

[4] No UPPAAL o símbolo [] representa o operador $\Box$ e o operador $<>$ é $\Diamond$.

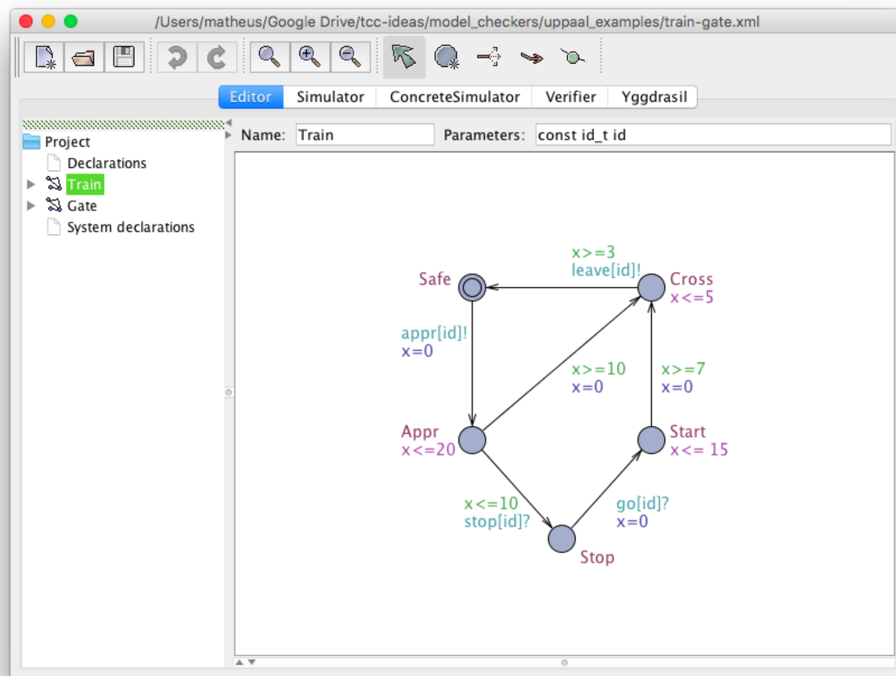



**Figura 13 – Aba de edição de modelos no UPPAAL**

**Fonte: Autoria Própria**

/Users/matheus/Google Drive/tcc-ideas/model_checkers/uppaal_examples/train-gate.xml

Editor | Simulator | ConcreteSimulator | Verifier | Yggdrasil

Overview

```
Train(1).Appr --> Train(1).Cross
A[] Gate.list[N-0] == 0
A[] not deadlock
A[] Gate.list[N-1] == 0
A[] Train(1).Cross+Train(2).Cross+Train(3).Cross+Train(4).Cross <= 1
E<> Train(1).Cross && Train(2).Stop && Train(3).Stop && Train(4).Stop
E<> Train(1).Cross and Train(2).Stop
E<> Train(1).Cross
E<> Gate.Occ
```

Check

Insert

Remove

Comments

Query

```
E<> Train(1).Cross && Train(2).Stop && Train(3).Stop && Train(4).Stop
```

Comment

**Figura 14 – Aba de especificação de propriedades no UPPAAL**

**Fonte: Autoria Própria**

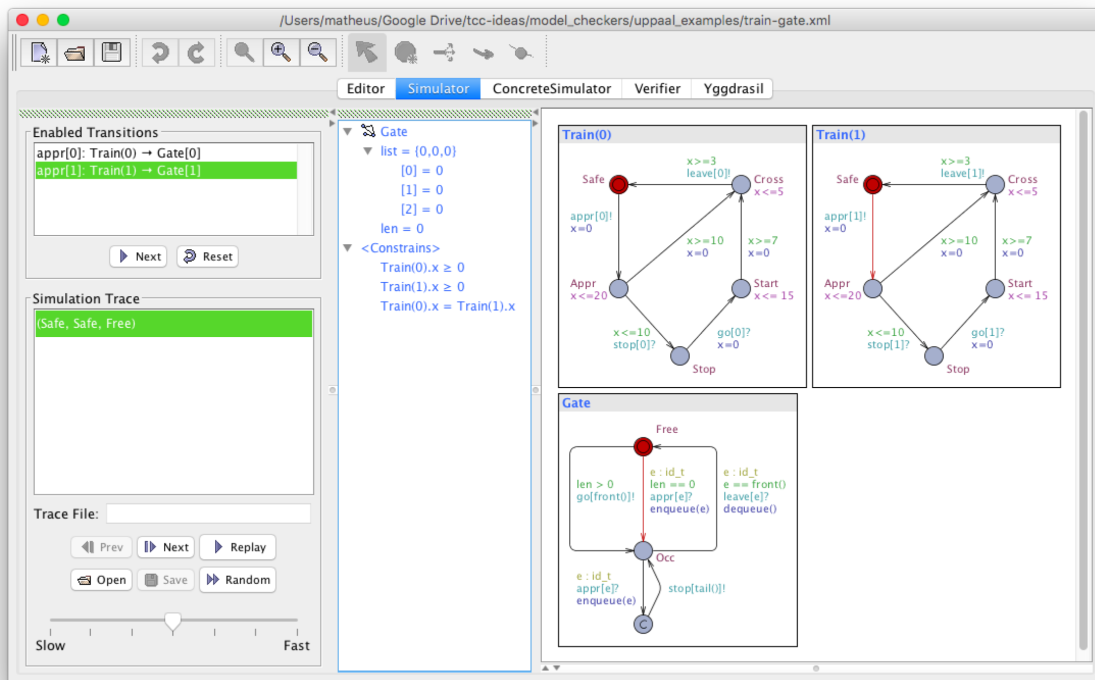



Figura 15 – Aba de simulação de modelos no UPPAAL

Fonte: Autoria Própria

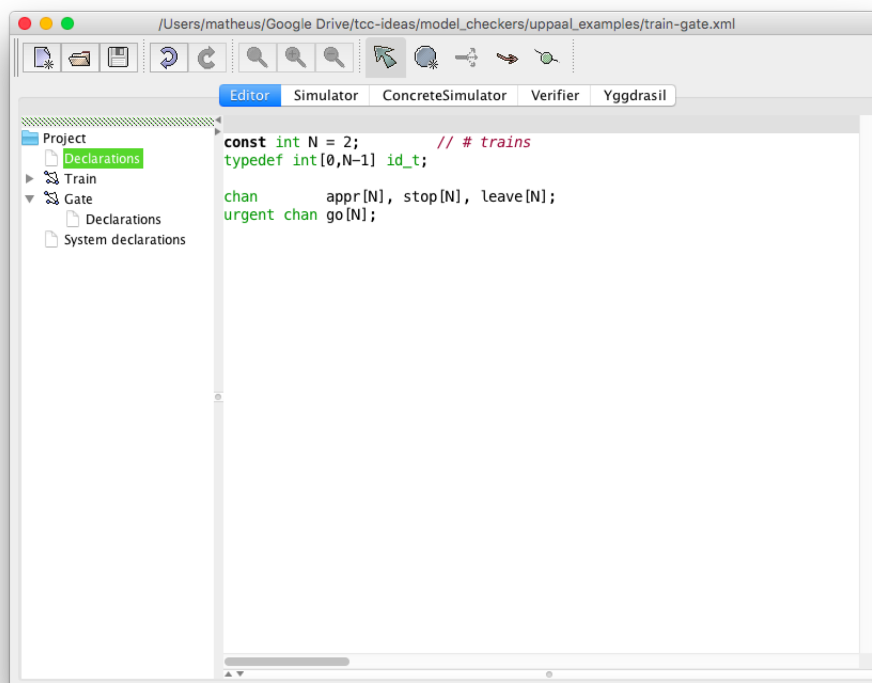



Figura 16 – Aba de declaração de variáveis, constantes e canais no UPPAAL

Fonte: Autoria Própria

O UPPAAL utiliza terminologia própria para *locations* (representadas por círculos). *Locations* podem ser marcadas como: *committed* para indicar que o tempo não pode passar e uma transição de *locations committed* (não há *interleaving*) deve ser executada imediatamente (visualmente adiciona-se um *C* dentro da *location*); *urgent* para indicar também que o tempo não pode passar (é acrescentado um $\cup$ dentro da *location*) e como *initial* para denotar a *location* inicial (círculo interno dentro da *location*) para indicação visual. Além disto, é possível adicionar *invariantes* as *locations* para especificar uma condição limite de permanência nas mesmas, por exemplo, que o tempo passado não pode ser mais que *x* unidades de tempo.

***Exemplo***

Um sistema de cruzamento ferroviárioa é apresentado como exemplo para ilustrar alguns dos conceitos aqui discutidos. O modelo encontra-se no trabalho de Yi, Pettersson e Daniels (1994) e está livremente disponível no repositório[5] de modelos do UPPAAL. Trens de vários trilhos precisam atravessar uma ponte, a qual é equipada com um controlador que recebe sinais vindos dos trens que irão atravessá-la. O objetivo é apresentar um modelo que possui capacidades temporais e garante que apenas um trem esteja cruzando a ponte em um dado momento. A Figura 17a mostra o modelo do trem e a Figura 17b modela o controlador. As *locations* são descritas a seguir:

**safe** um trem está distante por um período de tempo indefinido. Ao aproximar-se, *appr[id]!*, o trem comunica-se com o controlador a fim de que os passos necessários para o fechamento dos portões e/ou a parada do trem que se aproxima, caso haja outros trens já esperando para cruzar a ponte. Observe que cada trem possui um identificador *id* e um relógio $x$.

**appr** um trem está se aproximando não levando mais que vinte unidades de tempo ($x \leq 20$) para chegar até a ponte. Um sinal de parada só pode ser enviado em, no máximo, dez unidades de tempo.

**stop** um trem recebe um sinal para esperar, *stop[id?]*, ficando nesta *location* até que receba o sinal para prosseguir, *go[id]?*. Note que para a transição ser executada o valor de *x* deverá ser igual ou menor que dez.

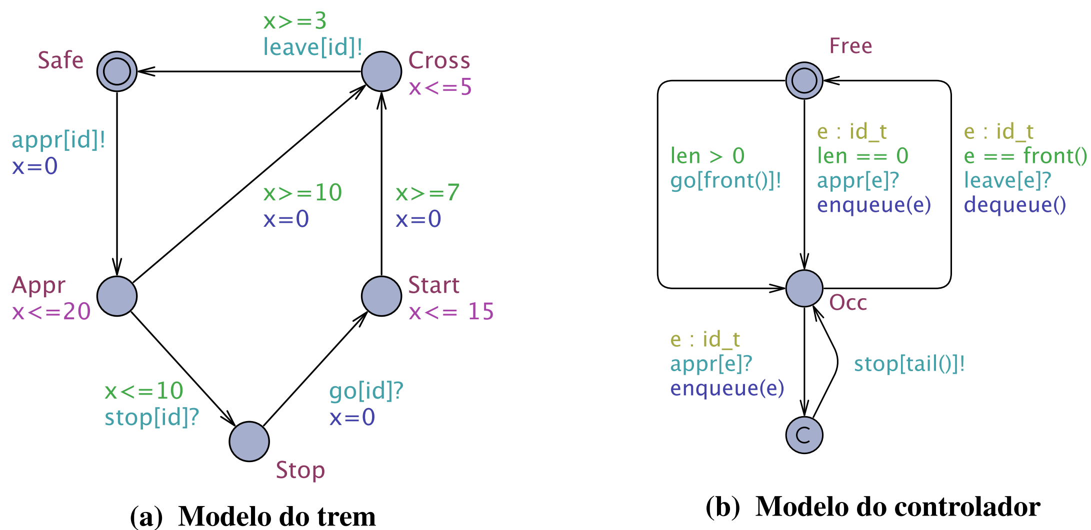
**start** um trem está se preparando para sair, o que leva entre sete e quinze unidades de tempo (condição $x \geq 7$ sobre a transição e $x \leq 15$ na *location*).

**cross** um trem está se prepara para cruzar a ponte, o que leva entre três e cinco unidades de tempo para acontecer (condição $x \geq 3$ na transição e $x \leq 5$ na *location*). Tendo atravessado-a, o trem envia um sinal para o controlador, *leave[id]!*, informando que atravessou com sucesso.

[5] Acessível a partir do UPPAAL.

**free** nenhum trem está esperando para atravessar a ponte. A *location* sincroniza em *appr[id]?* criando uma variável (*e : id_t*) para identificar o trem que está vindo e enfileira essa variável criada, *enqueue(e)*. Note que esta transição pode ser tomada somente se a fila estiver vazia (*len == 0*). Caso contrário, o controladoe sinaliza o trem na cabeça da fila, *go[front()]!*, para que deixe já que o último trem cruzou com sucesso a ponte.

**occ** o controladoe está esperando um trem sair. Se o trem que está vindo sinaliza que está se aproximando, *appr[e]?*, a variável *e* identificando esse trem é enfileirada e o trem recebe um sinal para parar, *stop[tail()]!*. A *location* é mudada pra *free* quando sincroniza sobre o canal *leave* e desenfileira aquele trem da lista pela função *dequeue()*. Quando um trem termina de cruzar a ponte e é o que estava no início da fila, *e == front()*, ele sincroniza em *leave[e]?* sendo o operação *dequeue()* executada para retirar da fila a variável que representa o trem. A *location* marcada com *C* significa que é uma *location committed* na qual o tempo não pode progredir; assim, uma transição deve ocorrer imediatamente.




**(a) Modelo do trem** **(b) Modelo do controlador**

**Figura 17 – Modelo de um simples sistema ferroviário no UPPAAL**

**Fonte: Repositório de modelos do UPPAAL**

***Propriedades***

Algumas propriedades para o modelo acima são (BEHRMANN; DAVID; LARSEN, 2006):

- E <> Train(1).Cross ∧ Train(2).Stop ∧ Train(3).Stop ∧ Train(4).Stop (*reachability*)
  Esta propriedade diz que se o Train1 está atravessando a ponte todos os outros trens estão parados até que o Train1 tenha cruzado a ponte com sucesso. Essa mesma propriedade pode ser especificada para os outros trens.

- A[] Train(1).Cross+Train(2).Cross+Train(3).Cross+Train(4).Cross $\leq 1$ (*segurança*)
  Em todos os estados possíveis somente um trem ($\leq 1$) pode cruzar a ponte por vez.

- Train(1).Appr – –> Train(1).Cross (*leads to*)
  Uma vez que Train1 sinaliza (o controlador da ponte) que está se aproximando, em algum momento irá atravessar a ponte.

- A[] not deadlock (*segurança*)
  Prova que o sistema está livre de *deadlock*.

Apesar de simples, essas propriedades são valiosas para garantir que um sistema cumpra de fato aquilo que deva fazer e mostram-se importantes já que a verificação de erros de modo manual é difícil e passível de erros.

## 3.4 UPPAAL SMC

A despeito de implementar algoritmos eficientes, o UPPAAL apresenta limitações na realização de *model checking*, particularmente em consequência do problema da explosão de estados. Ademais, há sistemas que exibem compormento probabilístico. Para tais fins, o UPPAAL possui uma versão probabilística que analisa propriedades dentro de um intervalo de confiança, muito semelhante ao PRISM. A versão é denominada UPPAAL *Statistical Model Checking* (SMC) e é apresentada em David *et al.* (2015) e já é distribuída juntamente com o UPPAAL. O formalismo utilizado nesta extensão é descomplicado, simples, e possibilita maior expressividade do que as lógicas temporais convencionais.

Para a especificação de propriedades é utilizada a *Metric Interval Temporal Logic* (MITL) com peso. Os tipos são estimativa de probabilidade e teste de hipóteses. Na primeira é calculada a probabilidade de uma dada propriedade especificada ser válida no sistema (similar ao modo quantitativo do PRISM). Na segunda uma probabilidade limite é informada pelo usuário e o sistema informa se a hipótese é verdadeira ou não (similar a asserções no PRISM).

## 3.5 DIVINE

Durante a realização deste trabalho foi encontrado um *model checker multicore* chamado DIVINE. O UPPAAL, apesar de ser robusto sendo continuamente desenvolvido ao longo dos anos, ainda não implementa algoritmos de verificação paralelos. Por usar somente um processador, o UPPAAL tende a demorar muito mais do que se usasse uma abordagem *multicore*.

DIVINE (BARNAT *et al.*, 2013) é um avançado *model checker multicore* que aceita como entrada modelos elaborados no UPPAAL verificando se exibem ou não *(time) deadlock*.

Aceita ainda como entrada LLVM de programas C e C++.

## 3.6 COMPARAÇÃO ENTRE *MODEL CHECKERS*

As ferramentas descritas neste capítulo foram SPIN, PRISM, UPPAAL e UPPAAL SMC. O Quadro 2 resume as informações descritas nas seções anteriores sendo subdividida em duas colunas principais: *Geral*, onde informações sobre o ambiente de operação e disponibilidade são dadas; e *Model Checking*, onde os pontos técnicos relacionados ao *model checking* são abordados, tais como linguagem utilizada na especificação de propriedades e tipo da especificação de modelos. Os pontos principais da tabela para o presente trabalho são: o tipo de especificação do modelo já que é preferível uma representação gráfica — protocolos de comunicação apresentam facilidade maior de serem modelados através de estados; a linguagem para a especificação de propriedades, pois espera-se que sejam baseadas nas lógicas temporais comumente abordadas na literatura; e os cenários de aplicação focando-se em protocolo de comunicação.

| **Model Checker** | **Geral** | | | **Model Checking** | | | | |
|---|---|---|---|---|---|---|---|---|
| | Distribuição | GUI | Plataformas | Modelo | Linguagem | *Query Language* | Tipos de MC | Aplicações |
| Spin | *open source* | iSpin | Linux, MAC, Windows | textual | PROMELA | LTL | não probabilístico | Geral |
| PRISM | | xprism | | | Linguagem PRISM | LTL, PLTL, CTL, PCTL | probabilístico | InfoSec, Biologia, Protocolos de Comunicação e mais |
| UPPAAL | *free* | sim | | gráfico | - | (subconjunto da) TCTL | não probabilístico | RTSs, Protocolos de Comunicação |
| UPPAAL SMC | | | | | | MITL | probabilístico | |

**Quadro 2 – Comparação entre *model checkers***

**Fonte: Autoria própria**

Nota-se que o SPIN, embora amplamente aplicável, não dispõe de variáveis relógio e, por isso, não é utilizado na modelagem de sistemas onde o tempo tem um papel fundamental, principalmente nas propriedades. O PRISM tem sido aplicado em várias áreas incluindo *InfoSec* (*Information Security*) e Biologia. O UPPAAL tem emergido nos últimos anos como um dos *model checkers* de destaque sendo aplicado especialmente na verificação onde o tempo é crítico para o sistema. Sua extensão, UPPAAL SMC, por ser uma adição recentemente (2011) integrada ao UPPAAL, ainda não tem sido usada em muitos trabalhos.

Para a realização do presente trabalho optou-se pelo UPPAAL pelas seguintes razões: (*i*) em Rosas (2014) foi utilizado o UPPAAL como ferramenta para verificação formal do CSMA/CA do padrão 802.11; (*ii*) especificação de modelos graficamente, o que facilita o processo de construção do modelo; (*iii*) por ser adequado para protocolos de comunicação de redes; (*iv*) e por possibilitar a criação de modelos através de uma interface gráfica o que é desejável para a elaboração deste trabalho.

# 4 PROTOCOLOS MAC PARA REDES SEM FIO

Este capítulo tem como objetivo explicar o padrão de controle de acesso ao meio atualmente utilizado em redes locais sem fio (*half duplex*) e uma proposta alternativa cujo projeto assumo rádios *full duplex*. Para tanto contém duas subseções. A Seção 4.1 explica como o processo de acesso ao meio é realizado em redes 802.11. A Seção 4.2 aborda um protocolo para redes *full duplex*.

## 4.1 PROTOCOLO MAC PARA REDES *HALF DUPLEX* SEM FIO

Evitar perda de *frames* aumenta a vazão da rede e diminui o tempo de resposta, o que afeta diretamente a Qualidade de Serviço (*Quality of Service*) de uma rede. Protocolos de acesso acesso ao meio são empregados para lidarem com tais problemas. No padrão estabelecido pelo *Institute of Electrical and Electronics Engineers* (IEEE) em IEEE Computer Society, LAN/MAN Standards Committee e Institute of Electrical and Electronics Engineers (2007), é definido um protocolo de prevenção de colisão em redes sem fio IEEE 802.11. O protocolo especifica uma função de acesso ao meio denominada Função de Coordenação Distribuída (*Distribution Coordination Function*, DCF), a qual coordena o acesso das estações ao meio.

Uma *mobile station* — abreviada por STA em IEEE Computer Society, LAN/MAN Standards Committee e Institute of Electrical and Electronics Engineers (2007) — faz uso do protocolo *Carrier Sense Multiple Access / Collision Avoidance* (CSMA/CA) para verifica se o canal compartilhado está ocioso ou não. Caso ocioso por um período de tempo específico, pode iniciar uma transmissão. Caso contrário, a mesma precisa esperar um período de tempo antes de dar início a uma transmissão (Subseção 4.1.1). O período neste caso é dado pela espera até o término da transmissão mais um *DCF Interframe Space* (DIFS) mais o tempo de *backoff* (Subseção 4.1.2) da estação. O processo de *backoff* é interrompido se uma transmissão ocorrer enquanto o meio estiver ocioso e retomado somente quando estiver livre.

Os intervalos são estabelecidos[1] através de diferentes valores de tempo denominados *Interframe Space* dentre os quais destaca-se o DIFS e o *Short Interframe Space* (SIFS). O SIFS estabelece um curto intervalo de tempo para priorizar o envio de um *acknowledgement*, isto é, um *frame* de controle que confirma uma recepção com sucesso de um *frame de dados recém enviados*. O processo RTS/CTS (explicado adiante na Subseção 4.1.4) também respeita este período de tempo. Assim, destaca-se que nestes dois tipos de transmissão o intervalo SIFS sempre é respeitado, isto é, é preciso sempre esperar o seu término. DIFS é o tempo de espera por uma STA antes de iniciar uma transmissão contanto que, durante todo o passar do tempo do DIFS,

[1] Todos os valores são conforme definidos em IEEE Computer Society, LAN/MAN Standards Committee e Institute of Electrical and Electronics Engineers (2007) do contrário explicitamente especificados.

o meio seja detectado como ocioso; seu valor é maior do que o SIFS para que haja prioridade para alguns tipos de comunicações. A unidade de tempo adotada para o SIFS e o DIFS é o microssegundo.

A Subseção 4.1.1 explica com mais detalhes a DCF. A Subseção 4.1.2 explana o *backoff*, seguida pela Subseção 4.1.3 que discorre sobre o processo de ACK. A Subseção 4.1.4 finaliza descrevendo o mecanismo RTS/CTS adotado no padrão.

### 4.1.1 DCF

A DCF coordena o tempo de acesso em que cada estação irá acessar o meio. O processo é descrito a seguir, o qual é representado intuitivamente em Hammal *et al.* (2014, p. 2):

1. Antes de iniciar uma transmissão é necessário que uma *mobile station* (STA) espere por um DIFS. O meio é continuamente detectado durante este período, no qual deve estar sempre livre para que comece sua transmissão imediatamente após DIFS.
2. Se, após a passagem de DIFS, o meio estiver livre, a transmissão inicia-se logo em seguida.
3. Se, em qualquer momento durante o passar do DIFS, o meio for detectado como ocupado, ou quando, de início, este é detectado como ocupado, executa-se o seguinte procedimento:
    a) É necessário esperar que o meio esteja livre por um DIFS. Após o término deste, tem-se o início do processo de *backoff*. Caso o tempo do *backoff* seja diferente de zero, não se necessita calcular o valor do *backoff*, isto é, o processo foi iniciado anteriormente.
    b) Espera-se, então, a passagem de tempo de um *slot* (valor menor que um DIFS). Caso o meio esteja livre até o término do *slot*, o tempo do *backoff* é decrementado pelo valor de um *slot*. Caso, em qualquer instante, o meio for detectado como ocupado, suspende-se o *backoff*. Seu valor não é decrementado e é retomado após um período de um DIFS sem interrupções (ou seja, retorna-se para o item 3a).
    c) Quando o tempo do *backoff* chega a zero a STA começa a transmissão.

### 4.1.2 *Backoff*

*Backoff* é uma função utilizada para atrasar transmissões evitando que estações transmitam no mesmo tempo. No padrão IEEE 802.11 é utilizado um *backoff* exponencial. O cálculo do intervalo de *backoff* (microssegundos) é obtido por intermédio da equação 4.1.

$$\text{backoffTime} = Random(0, CW) \times aSlotTime \qquad (4.1)$$

onde $aCWmin \leq CW \leq aCWmax$. Os parâmetros *aCWmin*, *aSlotTime* e *aCWmax* são valores definidos pela camada física.[2]. O valor inicial para $CW$ é $aCWmin$. A função *Random()* retorna um (pseudo) número aleatório dentro de intervalo dado por dois valores que marcam o início e o fim do intervalo (inclusive). O processo de espera pelo meio deve repetir-se até que a STA consiga aferir o meio como ocioso.

O processo de *backoff* é chamado quando uma STA não percebe o canal ocioso durante o intervalo DIFS ou o *frame* de *acknowledgement* não é recebido dentro do intervalo *ACKTimeout* (Subseção 4.1.3). Há outros cenários em que este procedimento é usado mas isso está fora do escopo deste trabalho. A Figura 18 ilustra o processo de *backoff*.

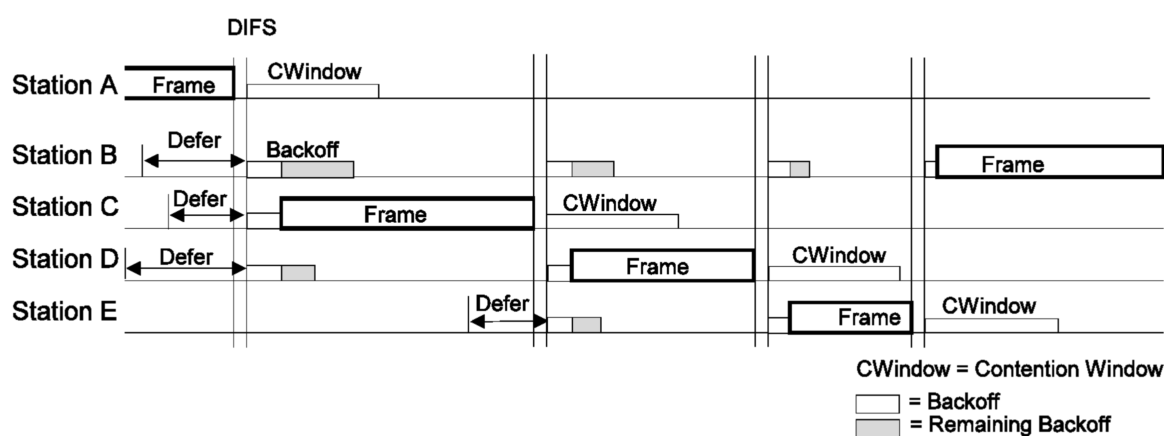



**Figura 18 – Processo de *backoff***

**Fonte: IEEE Computer Society, LAN/MAN Standards Committee e Institute of Electrical and Electronics Engineers (2007, p. 262)**

### 4.1.3 Processo de Confirmação de Transmissão

Após a finalização da transmissão de um *frame* de dados, a STA receptora confirma que recebeu os dados corretamente da STA remetente através do envio de um *acknowledgement (ACK) frame*. O processo é detalhado a seguir:

1. Após o término da transmissão, a STA remetente *A* inicia um contador de espera pelo ACK chamado *ACKTimeout*.

   a) Caso *A* receba o *ACK* da STA receptora *B* antes de *ACKTimeout* expirar, a transmissão foi realizada com sucesso e ambas as STAs retomam o processo de disputa pelo meio para enviar outros *frames*.

[2] Os valores dependem da camada física utilizada.

b) Caso ocorra a expiração do *ACKTimeout*, a STA *A* assume colisão e espera o meio ficar ocioso por DIFS antes de atualizar o valor da janela de contenção e tentar uma nova transmissão. O valor de $CW$ é atualizado conforme a equação 4.2.

2. A STA *B*, em seguida o término da transmissão de *A*, espera por SIFS e envia *ACK* para a STA *A*. *B* retoma então o processo de disputa pelo meio após sensoriar o meio como ocioso por DIFS.

3. A STA *A* recebe o *ACK* indicando uma transmissão realizada com sucesso e repete todo o processo de disputa pelo meio antes de estabelecer uma nova transmissão reinicializando o valor de $CW$ para $aCWmin$.

$$\text{CW} = (2 \times CW) + 1 \tag{4.2}$$

### 4.1.4 Mecanismo de Sensoriamento RTS/CTS

O *Request to Send / Clear to Send* (RTS/CTS) é mecanismo de sensoriamento virtual de canal cujo objetivo é evitar colisões causadas pelo problema do terminal escondido (*hidden node problem*)[3] é a reserva temporária de um meio compartilhado. Este modelo é descrito a seguir e ilustrado na Figura 19:

1. Após o processo de contagem do valor de *backoff*, uma STA envia em *broadcast* uma requisição de envio, *Request to Send* (RTS), para uma STA destino. Este *frame* RTS especifica o tempo total de duração da troca de mensagens a fim de que as outras estações considerem este tempo antes de tentarem uma transmissão.

2. A STA remetente espera um intervalo de tempo (*CTSTimeout Interval*).

   a) Se, antes do fim deste período, a STA receber um *Clear to Send* (CTS), o procedimento foi realizado com sucesso e a transmissão de dados é iniciada.

   b) Senão, entra em processo de *backoff* logo após o fim.

3. Após receber uma RTS a STA destino responde com um sinal de "pode enviar" (CTS) em *broadcast*.

4. Respeitando-se o intervalo *CTSTimeout Interval* (item 2a) o meio é reservado com sucesso para a transmissão. Do contrário a STA remetente aciona o processo de *backoff* (item 2b).

[3] Suponha dois nós A e C, um fora do alcance do outro. A transmite para um nó B, o qual fica no alcance de ambos (entre A e B). Como C não pode detectar que A está transmitindo para B (e vice-versa), C envia para B causando uma colisão. Uma bela representação pode ser vista em <http://www.texample.net/tikz/examples/terminals/>.

Esta abordagem soluciona o problema do terminal escondido não obstante introduza *overhead*, diminuindo a eficiência da utilização do meio pelas outras STAs. Destaca-se também que, pelo *overhead* introduzido, este modelo de comunicação é posto em prática somente quando o tamanho do *frame* de dados ultrapassa um limiar estipulado no 802.11 (variável *dot11RTSThreshold*).

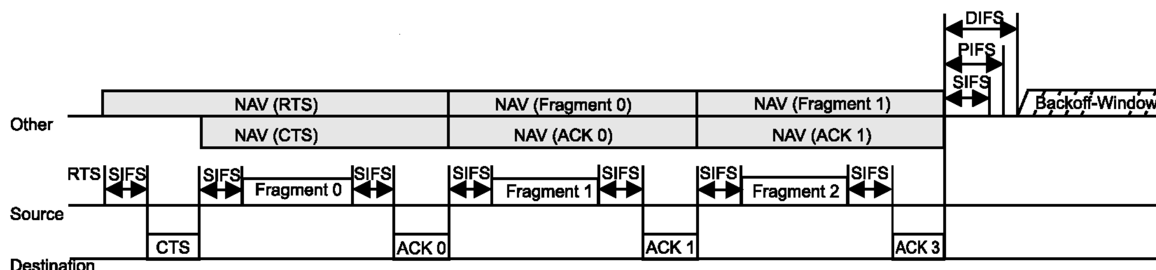



**Figura 19 – Mecanismo RTS/CTS**

**Fonte: IEEE Computer Society, LAN/MAN Standards Committee e Institute of Electrical and Electronics Engineers (2007, p. 266)**

## 4.2 PROTOCOLO MAC PARA REDES *FULL DUPLEX* SEM FIO

Protocolos *full duplex* possibilitam que duas estações aqui nomeadas $A$ e $B$ transmitam simultaneamente em um único meio sem fio compartilhado (JAIN *et al.*, 2011) o que, teoricamente, dobra a vazão da rede. Por ser recente, ainda não existe um padrão de protocolo MAC para redes *full duplex* sem fio. Esta seção pretende explicar um dos disponíveis[4], um protocolo CSMA/CA *full duplex* com cancelamento usando auto-interferência proposto em Jain *et al.* (2011). A seguinte terminologia é adotada pelos autores:

- ***primary sender*** STA que inicia uma transmissão primária.
- ***primary receiver*** STA destino de uma transmissão primária sendo também o *secondary sender*.
- ***secondary sender*** STA que inicia uma transmissão secundária.
- ***secondary receiver*** STA destino de uma transmissão secundária.

O controle de acesso ao meio é descrito a seguir:

1. *A* afere se o meio está disponível para uma transmissão (assim como no 802.11 padrão).

[4] Não nomeado pelos autores.

2. Se o meio estiver disponível *A* começa a sua transmissão. O processo de espera para o início de uma transmissão é o mesmo da DCF (ver Subseção 4.1.1).

   a) Após enviar o cabeçalho, *A* inicia um contador chamado *primary timer*.

   b) Se *A* receber de *B* um sinal dentro desse intervalo de tempo *A* continua transmitindo para B: uma transmissão *full duplex* foi estabelecida com sucesso.

   c) Se *A* não receber um sinal de *B* dentro desse intervalo, *A* assume que ocorreu uma colisão, atualiza sua janela de contenção e retransmite após DIFS e *backoff*.

3. Assim que recebe e decodifica o *header* do *frame* (para descobrir a estação remetente) B começa uma transmissão secundária enviando sua mensagem ou um *busy tone*[5] caso não tenha nenhum *frame* disponível (para evitar o problema do terminal escondido visto anteriormente).

   - Caso *B* tenha dados para enviar para *A* o tamanho do *frame* é escolhido de forma que a transmissão de *B* sempre termine ao mesmo tempo que a transmissão de *A*.[6]

4. A e B realizam um *ACK* simultâneo (mais detalhes na Seção 6.6).

Nos experimentos realizados pelos autores em (JAIN *et al.*, 2011), o protocolo traz ganhos expressivos, apesar de não dobrar a vazão da rede. Em geral, o aumento de vazão constatado foi de aproximadamente $45\,\%$. Assim, mesmo restritivo — somente um *primary receiver* pode iniciar uma transmissão secundária — o protocolo traz ganhos consideráveis às redes sem fio.

[5] Um *busy tone* é um sinal sem informação significativa enviado para um receptor para ocupar um meio de forma a prevenir que outros transmissores utilizem o meio.

[6] Essa é uma suposição adotada para a realização deste trabalho não sendo explicitamente especificada em Jain *et al.* (2011) (detalhes na Seção 6.2).

# 5 TRABALHOS RELACIONADOS

As seções sequentes discutem alguns trabalhos relacionados que auxiliam na compreensão do hodierno cenário da aplicação de métodos formais em protocolos de comunicação e, mais especificamente, em protocolos de rede. A Seção 5.1 aponta dois estudos que fornecem um panorama sobre os métodos formais aplicáveis em protocolos de comunicação. A Seção 5.2 relata a utilização de *model checking* na verificação formal de alguns protocolos de rede sem fio. Pretende-se, por intermédio deste estudo, delinear uma visão geral da aplicação de métodos formais na verificação de protocolos de comunicação, notadamente em protocolos de rede. A Seção 5.3 encerra o presente capítulo esquematizando os trabalhos relacionados a este.

## 5.1 MÉTODOS FORMAIS, PROTOCOLOS DE COMUNICAÇÃO E *NETWORKING*

Em (BABICH; DEOTTO, 2002) os autores investigam a aplicação de métodos formais na área de protocolos de comunicação argumentando que podem colaborar no processo de especificação, validação e verificação dos mesmos. Alegam ainda que informalidade e pragmatismo ainda se fazem presentes no desenvolvimento desses protocolos. Primeiro, especificações de protocolos de comunicação geralmente são feitas em linguagem natural o que, inevitavelmente, introduz ambiguidades nas especificações. Segundo, programas executando corretamente têm sido usados como prova de que uma dada implementação satisfaz os requisitos definidos na especificação. Assim, percebe-se que a aplicação de métodos formais não é disseminada, sobretudo pelo fato de que, na visão da comunidade de redes, o esforço de aplicar um método formal não justifica adequadamente a relação custo-benefício.

Retratada essa visão geral, os autores apontam alguns formalismos que podem apoiar o desenvolvimento de protocolos com rigor lógico dentre os quais destacam-se:

- **SDL** (*Specification and Description Language*) é uma linguagem gráfica intuitiva com rigor lógico baseada em máquinas de estados finitos (*Finite State Machines*, FSMs) para a especificação de protocolos de comunicação capaz de representar estados, sinais de entrada e saída, transições não determinísticas e mais. Suas principais aplicações são em sistemas de comunicação e de tempo real, e protocolos de comunicação. Modelos são executáveis em ferramentas que oferecem suporte.

- **Estelle** é uma linguagem formal textual utilizada na verificação formal de protocolos de comunicação. A especificação é dada em duas partes: declaração de todas as variáveis ou canais a serem utilizados no modelo seguida pela descrição do modelo. A verificação ocorre, então, dentro de um ambiente que forneça suporte para a escrita de especificações Estelle.

- **Redes de Petri** são modelos baseados em FSMs adequados para a especificação de protocolos de comunicação que possuem variantes aplicáveis em diferentes contextos. Um modelo é basicamente constituído de lugares, transições, e arestas direcionadas. É um formalismo usualmente aplicado em protocolos de comunicação e redes de telecomunicações.

Babich e Deotto (2002) concluem que métodos formais, apesar de possuírem um alto custo inicial, uma vez aprendidos, podem ser reusáveis no desenvolvimento de outros protocolos.

Observa-se que existe uma gama de métodos formais textuais ou gráficos disponíveis para aplicação e que estes são mais *amigáveis* que os textuais bem como são adequados para simulações tendo boas capacidades de modelagem. Essa facilidade de uso não implica que métodos formais textuais não são aplicáveis em determinados contextos mas refere-se ao uso inicial da ferramenta por parte de quem é inexperiente na área. Por exemplo, no *model checker* PRISM a modelagem é feita em linguagem textual mas que possui igual representação gráfica.

Destarte o trabalho de Babich e Deotto avalia concisamente alguns métodos formais disponíveis considerando conceitos como sintaxe e comunicação, além de apontar para outros trabalhos que tratam da aplicação dos métodos formais descritos. Exemplifica aplicações com o protocolo *Go-Back-N* ARQ (*Automatic Repeat Request*) no qual também o UPPAAL é utilizado para modelagem. Por alguma razão, contudo, exemplos de verificação de propriedades não são dados, fazendo com que os modelos sejam úteis somente na simulação e nada mais, não sendo, então, aplicado o *model checking*. Ainda segundo o trabalho de Babich e Deotto o uso do UPPAAL limita-se à verificação de aspectos relacionados ao tempo em protocolos de comunicação em tempo real. No entanto, desde a publicação do trabalho a ferramenta tem sido utilizada de fato na verificação de protocolos e não somente considerando aspectos específicos de valores de relógios como apresentado no trabalho.

Qadir e Hasan (2015) adotam uma posição similar à de Babich e Deotto defendendo que métodos formais deveriam ser aplicados em protocolos de rede. Afirmam que a comunidade de redes compartilha de uma cultura de desenvolvimento informal e pragmática. A razão está no fato de que a especificação comumente se dá por meio de RFCs (*Request for Comments*), as quais são descrições textuais para a especificação de protocolos. E, a despeito do constante crescimento do uso de métodos formais em outras áreas, o mesmo não vem ocorrendo nesta área importante.

Alguns exemplos da aplicação de métodos formais listados no artigo de Qadir e Hasan (2015) são: verificação de ausência de *deadlocks* e propriedades de segurança; detecção de loops e análise de atingibilidade; e segurança e verificação de redes. Verifica-se, então, que métodos formais encontram amplo uso na área de redes como um todo. Qadir e Hasan consideram ainda que o processo de *model checking* é relativamente de fácil aprendizagem e mais intuitivo quando comparado com outros métodos formais, o que contribui para sua maior aplicação na indústria. No entanto, ressaltam a presença do problema da explosão de estados e apresentam que técnicas como *model checking* simbólico ou probabilístico tem sido aplicadas para combater o mesmo.

Por fim, salientam que a aplicação de *model checking* é restrita a protocolos não sendo adequada para a verificação de programas devido à falta de estrutura sistemática do mesmo.

Observa-se, então, que para o presente trabalho o problema da explosão de estados, na casa dos bilhões para ausência de *deadlock*, pode limitar a verificação das propriedades a serem especificadas. Em vista disso, o modelo desenvolvido a partir do trabalho de Rosas (2014), contém algumas abstrações para viabilizar o processo de verificação formal (detalhes no Capítulo 6).

## 5.2 *MODEL CHECKING* E PROTOCOLOS DE REDE SEM FIO

Na Seção 5.1 foi discutida brevemente a aplicação de métodos formais na área de protocolos de comunicação. Agora, a presente seção limita-se a aplicação de *model checking* a protocolos de rede sem fio. Na Subseção 5.2.1 é discutida a aplicação de *model checking* probabilístico em redes de sensores. O protocolo de controle de acesso ao meio do padrão 802.11 é tratado na Subseção 5.2.2, onde uma verificação formal probabilística é apresentada. Uma variação deste protocolo é analisada na Subseção 5.2.3 seguida, então, pela Subseção 5.2.4 que examina o trabalho de Rosas (2014) que aborda o uso de *model checking* não probabilístico no CSMA/CA do padrão 802.11. A Subseção 5.2.5 aborda o mesmo protocolo verificado por Rosas (2014) mas com um componente a mais e uma configuração diferente para a DCF.

### 5.2.1 *Model Checking* Probabilístico Aplicado em Redes de Sensores

Em Fruth (2006) é apresentada uma verificação formal do protocolo 802.15.4 (voltado para redes de sensores sem fio) utilizando o *model checker* PRISM. Os dispositivos que participam nessa rede geralmente necessitam enviar poucos *bytes* de dados na comunicação possuindo baixo consumo de energia.

O CSMA/CA é utilizado somente para o envio de pacotes de dados sendo semelhante ao adotado no padrão 802.11 com algumas modificações. Para a modelagem é utilizado um autômato temporal probabilístico (*Probabilistic Timed Automata*, PTA). O modelo tem parâmetros configuráveis e foi elaborado com algumas suposições para torná-lo mais simples assim como para facilitar sua verificação (explosão de estados). Uma suposição presente, por exemplo, foi considerar o canal como sendo ideal (erros não ocorrem durante uma transmissão). Nota-se que parâmetros configuráveis possibilitam diferentes simulações. As propriedades foram especificadas em lógica de árvore de computação probabilística. Um exemplo de propriedade especificada é a probabilidade mínima de duas estações completarem com sucesso suas transmissões.

Os resultados obtidos apontam que, se o valor mínimo do expoente de *backoff* é incrementado, diminui-se tanto a probabilidade de colisão como o tempo esperado de transmissão. Outro resultado observado é que a transmissão em modo *slotted* aumenta a probabilidade de

transmissões com sucesso embora incorra num ligeiro aumento no tempo de transmissão.

O artigo dedica pouco espaço para a análise das probabilidades. Simplesmente são apresentados os resultados em uma tabela sem maiores discussões, o que poderia ressaltar de um modo melhor a importância das propriedades verificadas.

### 5.2.2 *Model Checking* Probabilístico e o Padrão 802.11 da IEEE

Apresentada a aplicação de model checking probabilístico no protocolo 802.15.4 na Subseção 5.2.1, esta subseção discute, de modo breve, a aplicação desse mesmo formalismo no padrão 802.11 da IEEE.

No trabalho de Kwiatkowska, Norman e Sproston (2002) *model checking* probabilístico foi utilizado na verificação formal do protocolo de controle de acesso ao meio utilizado no padrão IEEE 802.11. O modelo foi especificado e verificado no PRISM após sua elaboração e abstração no UPPAAL. O modelo conta com um número baixo — justificado pela explosão de estados — de estações, quatro no total, sendo duas remetentes e duas destino comunicando-se através de um canal. As propriedades verificadas foram:

- a probabilidade máxima do valor do contador de *backoff* de uma estação atingir um valor estipulado $k$. Observou-se que a medida que o valor de $k$ é incrementado a probabilidade do contador atingir esse mesmo valor diminui drasticamente. Isso mostra que as estações tendem a não entrar tanto no modo *backoff*.
- a probabilidade de uma estação enviar um pacote corretamente obedecendo um *deadline*. Como exemplo tem-se que, para um tempo limite de $10.000\,\mu s$, a probabilidade calculada é 0.9189, um valor razoável. A medida que o tempo limite foi aumentado, a probabilidade tendeu a um.

O número de propriedades verificadas foi limitado assim como foi o número de estações presentes no modelo. Isso deve-se, primariamente, ao problema da explosão de estados sempre presente no *model checking* e, secundariamente, ao fato de que o trabalho, realizado em 2002, foi desenvolvido em computadores de desempenho inferior aos disponíveis hoje. É importante notar que, além disso, as ferramentas utilizadas tem evoluído através do uso de algoritmos mais sofisticados e da melhoria de performance do código.

### 5.2.3 *Model Checking* e o Protocolo AMCLM

Após a explicação dos trabalhos anteriores sobre *model checking* aplicados em redes, esta subseção dedica-se a uma versão melhorada do CSMA/CA.

Em Ben-Othman, Mokdad e Bouam (2009) foi proposto o *Adaptive Multi-Services Cross-Layer MAC Protocol* (AMCLM) para redes 802.11 IEEE. Este protocolo visa aumentar a vazão da rede através do desligamento temporário de nós conectados à rede que estejam abaixo de um valor estipulado de interferência (*Signal to Noise Ratio* — SNR), o qual é uma razão entre a saída significativa de um sinal e a quantidade de ruído em *background*. Quanto maior esta razão, maior a interferência de um nó na rede. Assim, um limiar estipulado para nós móveis tende a aumentar a vazão já que os nós com maior interferência serão automaticamente desconectados. A motivação principal deste trabalho foi a *anomalia* presente em redes 802.11 (b). Conforme Heusse *et al.* (2003) em uma rede com vários nós, o de menor performance afeta os demais, de modo que, mesmo os nós com alta vazão exibirão valores abaixo do nó de menor capacidade de transmissão.

Hammal *et al.* (2014) modelou e verificou a DCF do AMCLM através do UPPAAL. Para a modelagem criou-se os modelos do funcionamento do CSMA/CA utilizando (um subconjunto da) máquina de estados da *Unified Modeling Language* (UML) e depois foram convertidos em autômatos temporais. Utilizou-se a UML para melhor captar o funcionamento do protocolo. Diagramou-se as estações e o *backoff* e, após a conversão, algumas propriedades foram verificadas: ausência de *deadlock* e uma estação em algum momento irá esperar por um ACK. Uma propriedade relacionada a esta última, entretanto, não pôde ser verificada (razão não apontada no texto), a qual especifica que a partir do estado de espera por um ACK volta-se ao estado inicial. Se verificada, tal propriedade representaria uma transmissão realizada com sucesso. Ademais, o modelo produzido assemelha-se bastante a uma solução para a verificação do padrão 802.11 em si. A diferença para o padrão é que um estado é acrescentado ao modelo para retratar a possibilidade de um nó ser desconectado ou conectado à rede.

Recorrente em modelos razoavelmente grandes, a explosão de estados não foi, em momento algum, citada em Hammal *et al.* (2014). Além disso, por algum motivo, não foi citado o número de estações utilizadas que, se em alto número, demonstraria a escalabilidade do modelo elaborado.

### 5.2.4 *Model Checking* e o Padrão 802.11 da IEEE, DCF

Esta subseção conclui a Seção 5.2 apresentando a verificação formal realizada anteriormente no Grupo de Pesquisa GEATC (Grupo de Estudos e Aplicações de Teoria da Computação) da UTFPR Campus Ponta Grossa.

Em Rosas (2014) o protocolo de controle de acesso ao meio (CSMA/CA) do padrão 802.11 IEEE foi modelado e formalmente verificado. Algumas suposições foram feitas a fim de facilitarem o processo de *model checking*, que são: supôs-se um canal perfeito, isto é, a transmissão é ideal (nunca há perda de mensagens); a quantidade de *frames* transmitidos é infinita; somente o modo de acesso *basic access* foi modelado. O modelo elaborado compreende três partes: (a) **troca de mensagens**: modela o envio e recebimento de mensagens; (b) ***acknowledgement***:

uma vez que uma mensagem é enviada precisa-se de um *ack* para confirmar se a transmissão ocorreu com sucesso ou não. Por ser diferente de uma transmissão comum foi modelado à parte; (c) ***backoff***: modela o processo de espera presente no protocolo. Verificou-se que o protocolo não possui *deadlock* para $n = 2$ (número de estações) e que é possível que um nó nunca transmita dentro de um intervalo de tempo especificado $T_n$, (verificada até $n = 4$). Intentou-se verificar a possibilidade de nunca ocorrer uma transmissão dentro de um intervalo $T_n$ porém, por causa da explosão de estados, não foi possível.

O trabalho fornece sólidos fundamentos para estudos futuros de variações do padrão, nomeadamente pelo modelo presente. Todavia, em consequência da explosão de estados, não foi possível encerrar um número razoável de estações nas propriedades verificadas, o que seria um cenário mais realístico do que duas ou quatro apenas. Ressalta-se ainda que, caso o modelo venha a ser utilizado como base para futuras modelagens mais complexas, é altamente provável que ocorra a explosão de estados para a propriedade de ausência de *deadlock*.

### 5.2.5 *Model Checking* o Padrão 802.11 da IEEE, DCF e PCF

Durante a realização deste deparou-se com o trabalho de Barboza *et al.* (2008), no qual *model checking* é usado para a modelagem e validação do padrão 802.11 da IEEE com uma componente extra, o modo PCF (*Point Coordination Function*), não presente em Rosas (2014).

Os autores, cientes de trabalhos relacionados à verificação do mesmo protocolo, argumentam que a validade do trabalho por eles desenvolvido é o fato de conter tanto a *Distributed Coordination Function* como a *Point Coordination Function* operando de forma integrada, algo até então ausente nos trabalhos publicados na literatura. Os autores defendem também que a elaboração do modelo abrange aspectos temporais[1] para a verificação de propriedades já que o tempo é crucial para serviços onde a QoS é importante. O *model checker* utilizado foi o UPPAAL. Ressalta-se que a implementação da função PCF é opcional por fabricantes de *hardware* de rede que implementam o protocolo de controle de acesso ao meio.

O modo de funcionamento PCF adota uma abordagem de controle de acesso ao meio centralizada em que uma estação, chamada *Point Coordinator*, após esperar um período de tempo específico (PIFS, PCF *Interframe Space*), assume a função de coordenar o acesso ao meio de todas as estações envolvidas. Logo, em vez de cada estação tentar o acesso ao meio individualmente (DCF), há uma estação coordenando esse processo. Além da função PCF, o funcionamento da DCF inclui o mecanismo RTS/CTS. Note que os modos podem ser executados alternadamente mas são mutuamente exlclusivos, em um dado momento somente um deles está em uso.

Para a modelagem da DCF, foram elaborados dois modelos, um representando o recebimento de mensagens e outro o envio de mensagens juntamente com o *backoff*. A PCF foi também modelada através de dois modelos, um para uma estação em modo PCF e outro para a

[1] No sentido de medição de tempo.

*Point Coordinator*. Há ainda outros dois modelos, um para modelar o sensoriamento do meio e outro modelando o meio físico.

O trabalho, por incluir uma componente a mais e mais configurações do padrão IEEE 802.11, apresenta uma grande quantidade de estados, o que resulta em um modelo complexo. Com a complexidade tem-se um modelo mais rico captando mais aspectos especificados em IEEE Computer Society, LAN/MAN Standards Committee e Institute of Electrical and Electronics Engineers (2007). Um aspecto positivo é o fato de os modelos desenvolvidos são modulares, facilitando o entendimento e expanão considerando trabalhos futuros.

O modelo, ou os modelos, tem oito propriedades especificadas e satisfeitas no total. As mais relevantes são a ausência de *deadlock*, que possivelmente uma estação permanecerá em colisão infinitamente. A probabilidade de colisão diminui consideravelmente a medida que a janela de contenção é aumentada. Uma outra propriedade verificada é o fato de que o funcionamento em PCF não interfere com o funcionamento em DCF, quando o PCF está ativo nenhuma estação pode acessar o meio como DCF. Ocorreu o probelma de explosão de estados quando mais estações foram incluídas na verificação das propriedades.

## 5.3 VISÃO GERAL DOS TRABALHOS RELACIONADOS

Apresentados alguns trabalhos de verificação formal de protocolos de rede, a Figura 20 esquematiza uma visão geral dos protocolos de controle de acesso ao meio diretamente relacionados a este (HAMMAL *et al.*, 2014; ROSAS, 2014; JAIN *et al.*, 2011), os quais servem como fundamento para a realização deste trabalho.

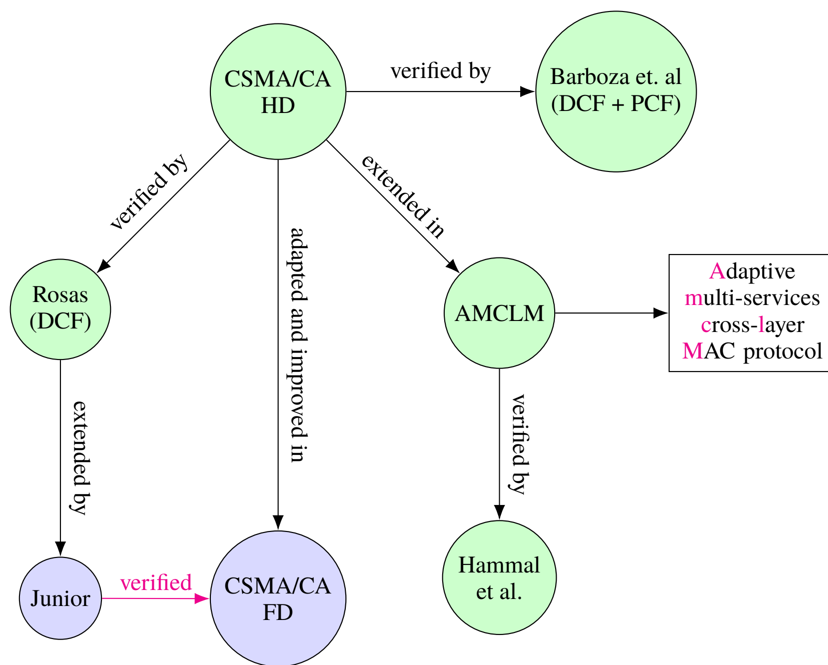



**Figura 20 – Trabalhos relacionados ao CSMA/CA HD e FD**

**Fonte: Autoria Própria**

# 6 DESENVOLVIMENTO

O presente capítulo descreve o modelo elaborado bem como algumas melhorias aplicadas no modelo de Rosas (2014). Primeiro a Seção 6.1 apresenta alguns padrões de modelagem utilizados. Em seguida a Seção 6.2 enumera as suposições adotadas no trabalho. A Seção 6.4 elucida o processo básico de transmissão de mensagens. Segue-se a Seção 6.5, a qual detalha o procedimento de *backoff* no modelo. Tem-se, em seguida, o detalhamento do processo de ACK na Seção 6.6, a qual é acompanhada pela Seção 6.7 descrevendo o mecanismo de tratamento de colisão bem como o modelo completo. A Seção 6.8 apresenta uma versão simplificada do modelo elaborado. Conclui-se com a Seção 6.9 listando algumas dificuldades deparadas durante a realização do trabalho. O modelo e os *scripts* de verificação estão disponíveis em <https://bitbucket.org/matheusjunior92/model-checking-802.11-fd>.[1]

## 6.1 PADRÕES PARA MODELAGEM

Em Behrmann, David e Larsen (2006) um conjunto de técnicas de modelagem são apresentadas para ajudar na elaboração de modelos relativamente mais simples, concisos e com quantidade de estados reduzida no UPPAAL. Dentre as técnicas discutidas, somente aquelas utilizadas neste trabalho são descritas a seguir.

- **Redução de variável** consiste em atribuir um valor específico para uma variável, geralmente zero, quando essa não está em uso, desconsiderando, assim, estados que sejam diferentes entre si apenas considerando o valor da variável, reduzindo portanto a quantidade de estados. O UPPAAL automaticamente aplica a redução de variável em variáveis relógio.

- **Passagem de valores síncrona** consiste em passar valores definidos no intervalo *MIN* e *MAX* (inclusive em ambos) para outros processos de modo síncrono. A ideia é utilizar um vetor de canais – *chan send[MAX - MIN + 1]* – onde em cada canal do vetor um valor específico é passado. O *receiver* então usa uma *feature* do UPPAAL chamada *select* para escolher em qual canal sincronizar. O canal escolhido terá um valor associado, sendo o valor então atribuído a variável local do processo. A Figura 21 ilustra o processo. A variável *random* é criada em uma das transições do *sender*. Um valor distinto é passado em cada canal do vetor de canais *send*. O *receiver* escolhe (escolha interna do UPPAAL) então um dos canais para sincronizar atribuindo o valor recebido a variável *value*.

- **Atomicidade** consiste em marcar estados como *committed*, assegurando atomicidade durante a verificação. Um estado *committed* é um estado no qual o tempo não passa. Além

[1] Tanto a declaração global de canais e variáveis e as declarações locais usadas por cada STA estão disponíveis.

disso, a próxima transição a ser tomada deve, necessariamente, ter origem neste mesmo estado, ou de qualquer outro estado também marcado como *committed*. Tem-se a redução então da quantidade de estados, ao menos que exista mais de um estado *committed*.

- **TIME unidades de tempo em uma *location*** Outro padrão é modelar a passagem de exatamente *TIME* unidades de tempo em uma dada *location*, ou seja, após esse tempo a transição é obrigatoriamente executada. Para tanto, é necessário definir uma invariante $x \leq TIME$ na *location* e também um *guard* $x == TIME$. Essa técnica está descrita no artigo, mas aparece em outra seção de Behrmann, David e Larsen (2006). Um exemplo é mostrado na Figura 22.

**(a)** ***Sender*** **(b)** ***Receiver***

**Figura 21 – Passagem de valores síncrona**

**Fonte: Repositório de modelos do UPPAAL**

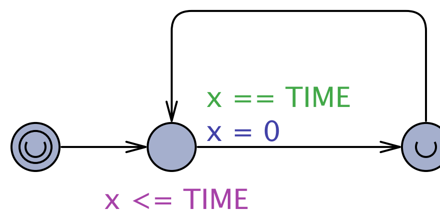



**Figura 22 – *TIME* unidades de tempo em uma *location***

**Fonte: Repositório de modelos do UPPAAL**

## 6.2 SUPOSIÇÕES

Algumas suposições foram realizadas neste trabalho durante e são:

- **ausência de *capture effect***: esse fenômeno refere-se ao fato de que, por placas de rede de diferentes fabricantes poderem ter potência de captação de sinal distintas, é possível que uma dada estação receba um pacote de dados enquanto outra não. Por ser um efeito não determinístico e complexo de modelar, tal fenômeno está fora do escopo deste trabalho.

- **tamanho fixo dos pacotes**: assumiu-se que os pacotes a serem enviados pelo *primary receiver* serão sempre menores do que os recebidos pelo mesmo. O tamanho é escolhido de forma que ambas as transmissões, primária e secundária, terminem ao mesmo tempo.[2] Esta suposição foi realizada já que o consenso na comunidade de redes é o uso de *busy tones* em protocolos *full duplex* MAC para garantir o término simultâneo das transmissões.

- **colisão**: ocorre quando uma STA *primary sender* não recebe um sinal do *primary receiver* em um intervalo definido por *primary timer* (quando duas ou mais STAs transmitem ao mesmo tempo), ou quando estações não recebem ACK.

- **terminologia**: para fins de explicação e conveniência, a terminologia canal primário e secundário será utilizada. O protocolo proposto em (JAIN *et al.*, 2011) não utiliza tal abordagem (é uma característica do protocolo *Piece-by-Piece* (PbP) proposto em Queiroz e Hexsel (2015)).

- **previsão de chegada de cabeçalho**: as STAs empregam um mecanismo de previsão de chegada de cabeçalho. Quando uma estação inicia o envio do cabeçalho, é possível as estações calcularem o tempo que levará para o seu envio fazendo com que o protocolo seja mais eficaz na utilização do meio.

## 6.3 ESTRUTURA DO MODELO

Uma simples esquematização do modelo elaborado neste trabalho pode ser visto na Figura 23, a qual tem como objetivo fornecer uma visão geral de como ele está organizado. Há basicamente quatro componentes presentes no modelo. O primeiro é o componente responsável pelo envio e recebimento de mensagens. O segundo lida com as colisões que venham a ocorrer. O terceiro tem como função a geração do valor de *backoff*. O quarto e último componente especifica o processo de transmissão de ACK. A ligação entre os componentes esquematiza suas relações de forma não estrita.

[2] É possível especificar o tamanho do *payload* de uma transmissão através do *script updateValues* que está disponível no repositório.

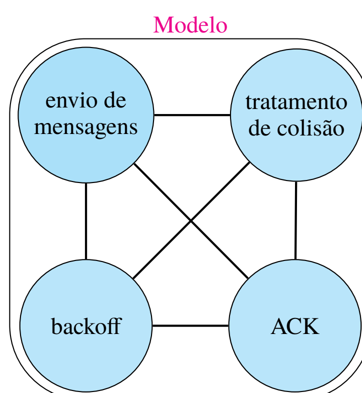



**Figura 23 – Relação dos componentes do modelo**

**Fonte: Autoria própria**

## 6.4 ENVIO DE MENSAGENS

O processo de envio de mensagens é modelado na Figura 24.

Todas as STAs começam na *location* inicial (marcada com um círculo interno), a qual também é marcada com ∪ (*urgent location*) para especificar que o tempo não deve passar e uma transição deve ser executada instantaneamente. Ao executarem a transição da *location* inicial para a *Sensing_for_DIFS*, cada STA executa a função *init_network()*, a qual preenche um vetor contendo os *ids* das STAs destinos de forma que cada STA não tenha, em seu vetor, seu próprio *id*, o que evita que uma STA envie uma mensagem para si mesma. Observe que o vetor é local para cada STA.

Ao chegar na *location Sensing_for_DIFS*, cada STA espera por DIFS antes de acessar o meio. Após esse tempo, uma STA, de modo não determinístico, acessa o meio – sincronizando com as outras STAs no canal *start_send* – dando início a uma transmissão, a qual é modelada através de duas *locations*, *Transmitting_Header_PTX* e *Transmitting_Data_PTX*. A primeira *location* modelo o envio do cabeçalho e o tempo passado nessa *location* é de $64\,\mu s$, um valor previsto no protocolo para envio do *cabeçalho*. A segunda *location* modela o envio de dados para a STA destino. Entre essas duas *locations* há uma outra, a *Waiting_Response*, que modela o temporizador de $75\,\mu s$ utilizado pela STA remetente. Não havendo nenhuma resposta da STA destino durante este período assume-se erro na transmissão e atualiza-se a janela de contenção.

A STA transmitindo o cabeçalho sincroniza com as outras no canal *send_header* indo para *Waiting_Response*. As outras STAs que estão em *Waiting_Medium_Free* também sincronizam indo para *Read_Header_PTX* onde verificam se o *header* é destinado à uma delas. Caso não seja, $dst\,! = id$, a STA espera o término da transmissão atual. Caso seja, a STA vai para a *location Reply_Header_PTX* onde decodifica o cabeçalho e envia-o para a STA que está transmitindo no

canal primário, o que dura cerca de 11 µs[3]. Observe que a transmissão secundária só é estabelecida caso o número atual de transmissões seja um (*nTx == 1* na *location Reply_Header_PTX*). Isso modela o fato de que uma colisão ocorre se duas ou mais STAs transmitirem ao mesmo tempo. Uma vez inicializada a transmissão secundária tem-se o estabelecimento de uma transmissão *full duplex*, uma STA está em *Transmitting_Data_PTX* e outra está em *Transmitting_Data_STX*. A duração da transmissão é dada por *TonlyData* que é igual a *Ts - Theader - TprimaryTimer*.

Ao término da transmissão primária, a STA sincroniza no canal *finish_sending* e vai para *Waiting_ACK_PTX* esperando lá por, no máximo, *ACKTimeout*. As outras STAs que sincronizaram vão para a *location Read_Destiny* onde verão que a mensagem não era endereçada para nenhuma delas. Percebe-se que há uma certa redundância, já que foi confirmado que a mensagem não era para nenhuma dessas STAs em *Waiting_Medium_Free*. Essa verificação é preservada pelo fato de que essa mesma *location* é utilizada posteriormente para a verificação do ACK.

[3] Decrementou-se 11 µs em uma unidade para evitar não determinismo entre continuar uma transmissão ou ocorrer colisão

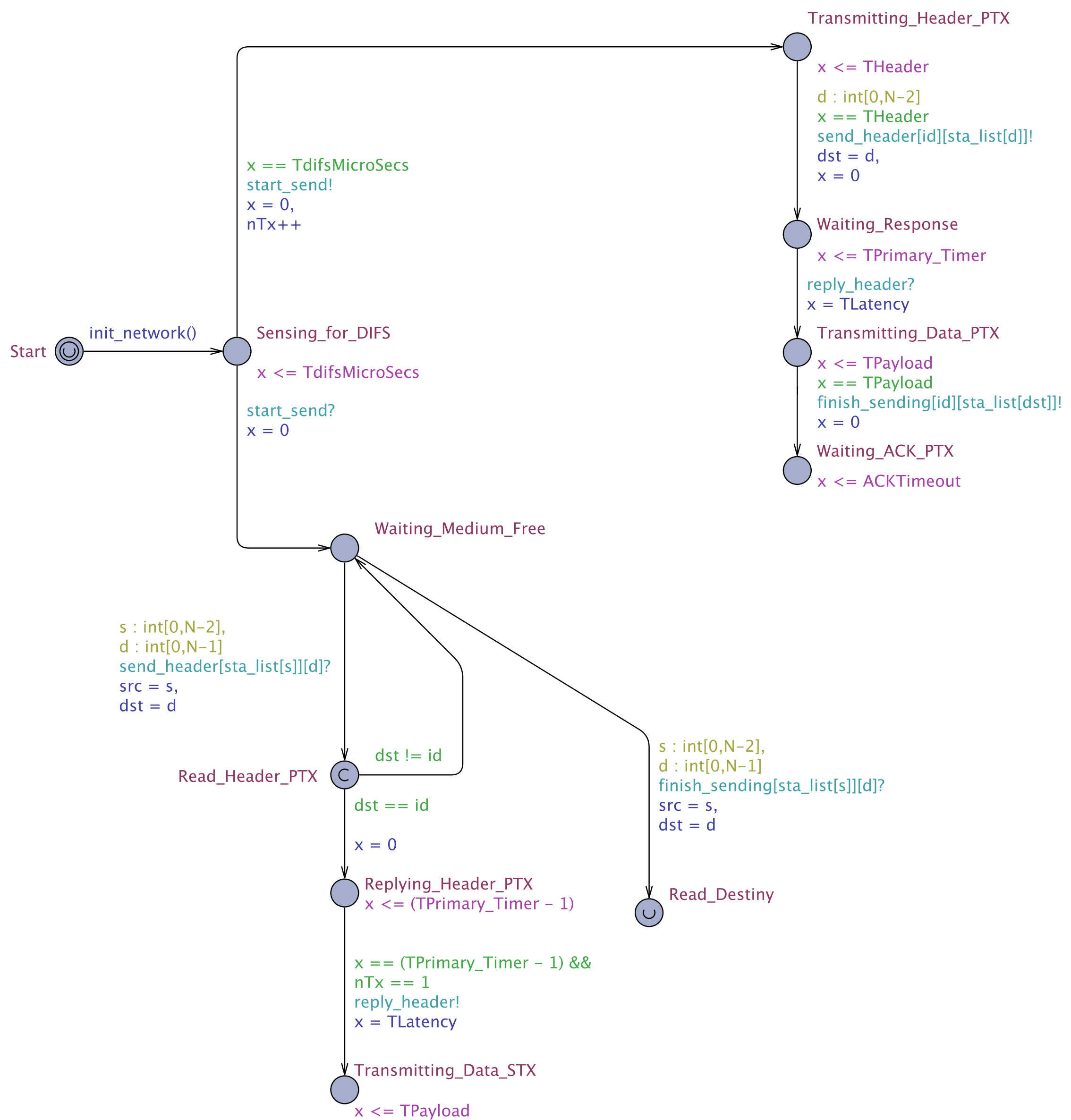



**Figura 24 – Envio de mensagens**

**Fonte: Autoria própria**

## 6.5 *BACKOFF*

O processo de *backoff* é apresentado na Figura 25.

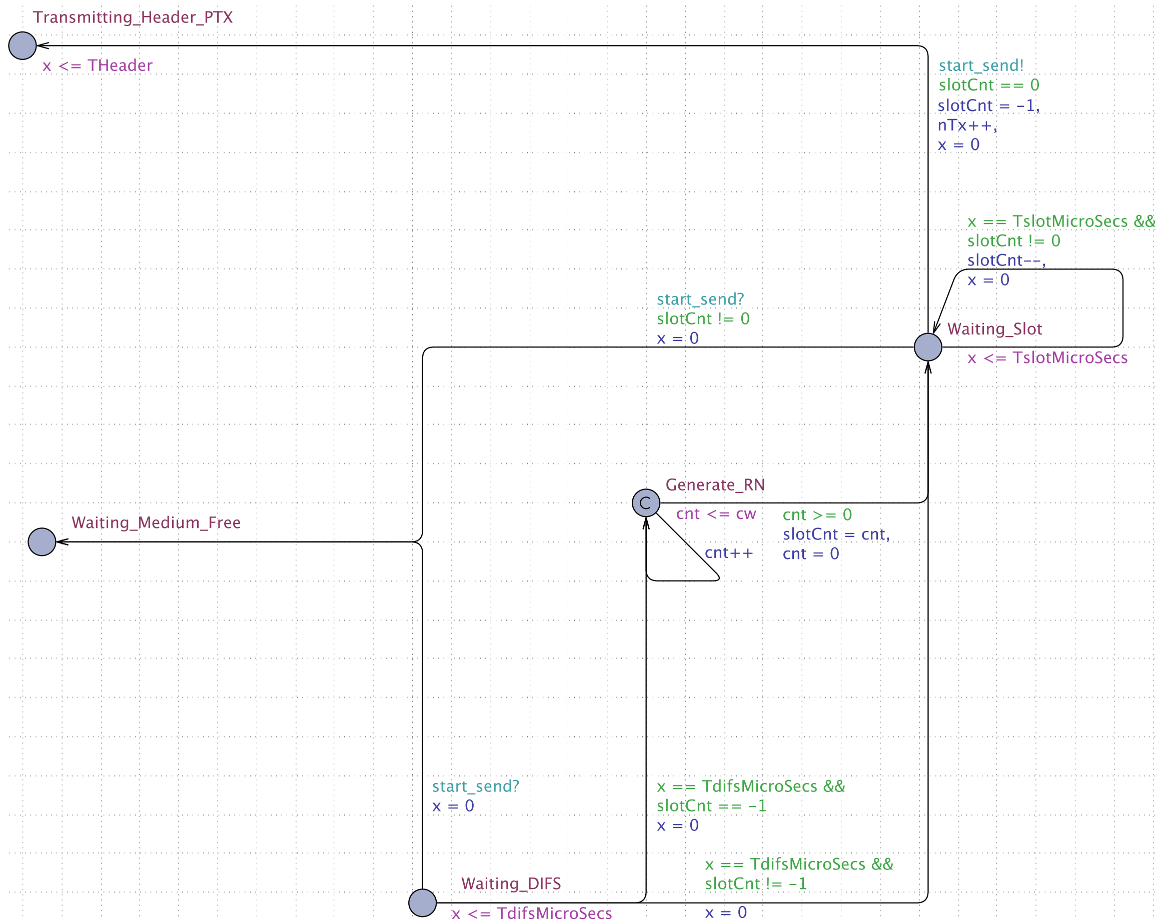



**Figura 25 – Processo de *backoff***

**Fonte: Autoria própria**

Após esperar DIFS ininterrupto (*Waiting_DIFS*), uma STA pode executar duas transições: uma vai para a *location Generating_RN* e a outra vai direto para *Waiting_Slot*. Na primeira é onde números aleatórios são gerados e é executada caso $slotCnt == -1$, isto é, a STA não iniciou o processo de *backoff* ainda. A segunda tem condição $slotCnt \neq -1$, ou seja, a STA já iniciou o processo de *backoff* em algum momento anterior. Essas duas transições representam o fato de que uma STA pode estar iniciando ou retomando o seu processo de *backoff*, respectivamente.

Em *Generating_RN* há duas transições habilitadas e como a escolha entre elas é não determinística, tem-se, então, um simples mecanismo de geração de números aleatórios não uniforme, ou seja, $cnt$ é incrementado de forma não determinística. A *location* é marcada como *committed* para que cada STA gere o seu número aleatório sem passagem de tempo e sem *interleaving*. A invariante $cnt \leq cw$ especifica que o número gerado deve estar entre o intervalo

$[0, CW]$ estabelecido no padrão IEEE.[4]

Após gerar o número aleatório, a STA atualiza a variável local *slotCnt* e muda para a *location Waiting_Slot*. Quando passar *TslotMicroSecs* (invariante $x \leq$ *TslotMicroSec* e *guard* $x ==$ *TSlotMicroSec*) e ainda houver *slots* de tempo (*slotCnt* $! = 0$), a variável local *slotCnt* é decrementada e o relógio local *x* é reinicializado com 0 para medir mais um *TslotMicroSecs*. Isso ocorre *slotCnt* vezes, deste modo a STA irá ficar por *slotCnt* $\times$ *TslotMicroSecs* nessa *location* caso não sincronize com outra STA (*start_send?*). Caso ocorra a sincronização, a STA passa a esperar o meio ficar livre novamente para retomar todo o processo de espera.

## 6.6 PROCESSO DE CONFIRMAÇÃO DE TRANSMISSÃO

O processo de ACK é descrito a seguir e modelado na Figura 26. Assim que **A** e **B** terminam suas transmissões ao mesmo tempo, o processo de ACK ocorre:

1. **A** começa a esperar pelo ACK de **B** (*Waiting_ACK_PTX*)[5] e **B** começa a esperar SIFS (*Waiting_SIFS_STX*).
2. Logo após SIFS, **B** começa a enviar o ACK no canal primário e **A** começa a receber o ACK – *Sending_ACK_PTX* e *Waiting_Medium_Free*. (Implícito: **A** também esperou SIFS; **A** começa a enviar ACK para **B**).
3. **B** termina a transmissão do ACK para **A** sincronizando no canal *finish_sending* e começa a esperar pelo ACK de **A** (*Waiting_ACK_STX*).
4. **A** confirma que recebeu o ACK de **B** (*Read_Destiny*), uma transmissão com sucesso.
5. **A** passa então a disputar o meio novamente (*Waiting_DIFS*) sincronizando com **B** no canal *send_stx_data*.
6. **B** lê o ACK de **A** (*Read_ACK_STX*), confirma que é o destinatário (*dst* $==$ *id*), outra transmissão com sucesso, e também começa a disputar o meio (*Waiting_DIFS*).

Quando **B** começa a enviar ACK (item 2), **A** implicitamente envia ACK para **B**, ou seja, não há um estado explícito modelando **A** transmitindo ACK para **B**. Assim, quando **B** termina de enviar ACK (item 3), **A** também termina de enviar seu ACK para **B**. Em seguida, **A** confirma o ACK de **B** (item 4), e **B** confirma o ACK de **A** (item 5 e item 6) – quando ocorre uma sincronização no UPPAAL o tempo não passa durante essa sincronização. Dessa forma, no modelo não é possível duas STAs estarem no estado *Waiting_ACK_PTX* e *Waiting_ACK_STX*.

[4] A Subseção 6.9.3 contém uma breve discussão sobre o mecanismo aqui utilizado.
[5] Entre parênteses tem-se o nome do estado no modelo.

Modelar explicitamente **A** e **B** transmitindo simultaneamente em paralelo aumentaria a quantidade de estados e transições, logo aumentando a complexidade do modelo. Assim, optou-se por abstrair **A** enviando ACK para **B**, o que não compromete o funcionamento do ACK simultâneo.

Ademais, como todo o processo de transmissão e os componentes envolvidos são tidos como perfeitos sem diferenças nas especificações técnicas de *hardware*, o único meio de ocorrer *ACKTimeout* para **A** é na ocorrência de colisões, ou seja, quando transmissões em paralelo acontecem. O mesmo não sucede com **B**, ou seja, há *ACKTimeout* para **B** mas ele nunca ocorre (no modelo). A razão é que, se duas STAs ou mais transmitirem em paralelo para **B**, esse não consegue entender os dados que estão chegando, com isso fica impossibilitado de iniciar uma transmissão paralela. Mas se **B** iniciar uma transmissão em paralelo, **B** conseguiu ler os dados corretamente, isto é, somente uma STA transmitiu para **B**. Como o meio é perfeito, o pacote de ACK não é perdido e *ACKTimeout* não ocorre.

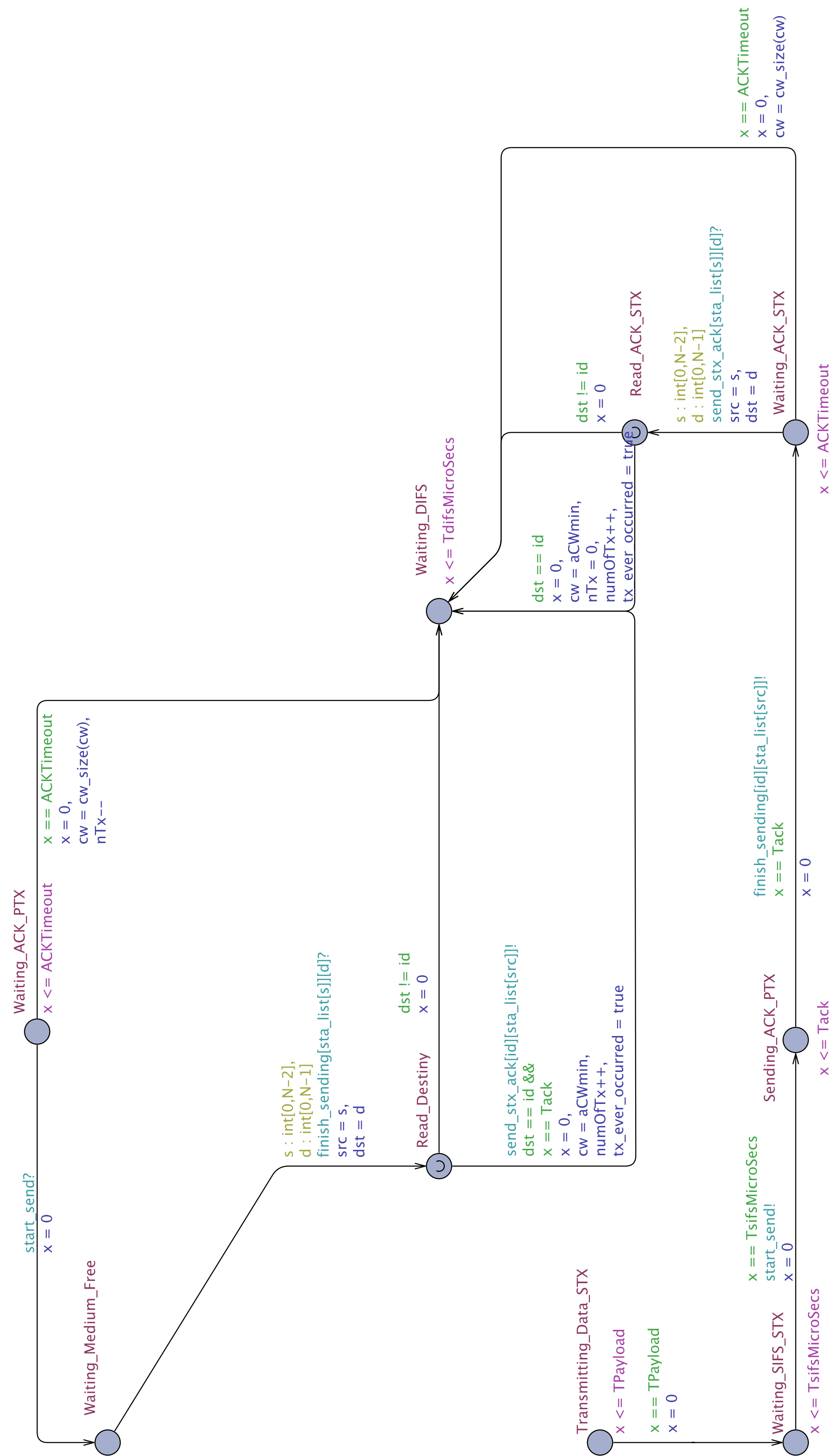



**Figura 26 – Processo de ACK**

**Fonte: Autoria própria**

## 6.7 TRATAMENTO DE COLISÃO

Quando uma STA sai da *location Waiting_Slot*, há duas transições possíveis. A primeira é iniciar uma transmissão primária e a outra é sobre o canal *start_send*.

No primeiro cenário a STA irá sincronizar sobre o canal *start_send*. Logo que iniciar sua transmissão, a STA indica que está transmitindo – a variável *nTx* é incrementada.

A segunda transição, sobre o canal *start_send*, acontece juntamente com o primeiro cenário visto anteriormente caso a(s) STA(s) que esteja(m) esperando pelo meio possua(m) mais do que um *slot* de tempo para decrementar. Essa transição modela o fato de que, quando uma STA começa a detectar o meio como ocupado, ela deve esperar o meio ser liberado. Essa sincronização também acontece em outras *locations* indo para *Waiting_Medium_Free*.

A variável *nTx* é utilizada no tratamento de colisão para controlar quantas STAs estão transmitindo simultaneamente. Se somente uma STA está transmitindo (*nTx == 1*), o processo de comunicação ocorre normalmente. Se *nTx* for maior que um, isso indica que mais de uma STA está transmitindo, o que leva a ocorrência de colisão, já que a STA destino não consegue ler os dados corretamente. *nTx* é decrementada quando ocorre *timeout*, seja do temporizador *primary timer* ou do *ACKTimeout*. *nTx* é atribuída com o valor zero caso uma transmissão tenha ocorrido com sucesso.

Observe ainda que quando ocorre *timeout* por falta de resposta da STA receptora, a sincronização sobre o canal *free_medium* ocorre. O meio é liberado já que todas as STAs envolvidas neste cenário não conseguem estabelecer nenhuma comunicação.

A Figura 27 mostra as arestas e *locations* nas quais a variável *nTx* é utilizada. O modelo completo pode ser visto na Figura 28 e sua documentação pode ser vista no Apêndice A.

**Figura 27 – Tratamento de colisão**

**Fonte: Autoria própria**

**Figura 28 – Modelo completo**

**Fonte: Autoria própria**

## 6.8 MODELO SIMPLIFICADO (SEM *PRIMARY TIMER* E *ACKTIMEOUT*)

Em uma outra versão do modelo elaborado a colisão foi tratada de modo diferente o que permitiu verificar a propriedade de *deadlock*. Nesse modelo simplificado é possível que uma STA execute uma transição que sincroniza no canal *interrupt_PTX*. Esse canal interrompe transmissões em andamento e foi criado para modelar o fato de que, se duas ou mais STAs transmitirem ao mesmo tempo, ocorre uma colisão por *ACKTimeout*, isto é, *ACKTimeout* expira e as STAs têm que reenviar suas mensagens. Assim, em vez de esperar todo o processo de transmissão ocorrer e *timeout*, o que traria um pouco mais de complicações para o modelo, assumiu-se uma colisão. As *locations* que sincronizam nesse canal são *Transmitting_Header_PTX* e *Waiting_Medium_Free*. No primeiro caso interrompe-se uma transmissão primária, assumindo diretamente uma colisão (transmissão em paralelo). Note que o valor da janela de contenção é alterado. No segundo caso, o canal comunica que o meio está livre às STAs que estão esperando por uma oportunidade de transmissão.

Com essa restrição de que não pode haver transmissões simultâneas a quantidade de estados a serem verificados diminui drasticamente tornando possível a verificação da propriedade de *deadlock* (ver Capítulo 7). Além disto, esse modelo simplificado não possui uma *location* para modelar o contador *primary timer*, isso porque não é possível ocorrer *ACKTimeout* já que transmissões em paralelo implicam no mecanismo de interrupção relatado anteriormente. Observe que, por introduzir essa suposição de assumir diretamente *backoff* em transmissões simultâneas, o modelo simplificado não está estritamente de acordo com a especificação do protocolo apresentado em Jain *et al.* (2011). Este modelo é representado na Figura 29.

**Figura 29 – Modelo simplificado (sem *primary timer* e ACKTimeout)**

**Fonte: Autoria própria**

## 6.9 APRIMORAMENTOS

No decorrer da execução deste trabalho foram constatadas algumas melhorias que poderiam ser aplicadas no modelo no modelo de Rosas (2014), as quais são expostas nas subseções a seguir. A Subseção 6.9.1 descreve uma melhoria relacionada com a não reinicialização de um relógio levando a uma diferença de prioridades para transmissões, descrita na Subseção 6.9.2. Após, a Subseção 6.9.3 considera a geração de números aleatórios. A Subseção 6.9.4 suscintamente apresenta um aprimoramento no processo de transmissão. Por fim, a Subseção 6.9.5 encerra comentando sobre a ausência de informações técnicas.

### 6.9.1 Relógio não Reinicializado

A variável relógio local $x$ não era reinicializada com o valor $0$ na *location Waiting_Slot*, o que fazia com que uma STA com *slotCnt* igual a 5, por exemplo, tivesse a mesma prioridade que outra STA com *slotCnt* igual a 1. Isso ocorre porque é possível que a variável *slotCnt* seja decrementada até 0. Há duas implicações por isso: uma STA com um valor baixo para *slotCnt* não necessariamente começa a transmitir primeiro que uma outra STA com um valor mais alto para a mesma variável. A outra implicação, contida implicitamente na anterior, é o fato de que uma STA com um valor alto de *slotCnt* pode começar a transmitir antes que outra STA com um valor menor para *slotCnt*. Percebe-se, então, que o modelo, neste quesito, não estava de acordo com o documento IEEE Computer Society, LAN/MAN Standards Committee e Institute of Electrical and Electronics Engineers (2007).

A solução encontrada: reinicializar $x$ com $0$ na *location Waiting_Slot*.

### 6.9.2 Ausência de Prioridade

Percebeu-se também que é possível que uma STA com um valor de *backoff* maior comece a transmitir primeiro que outra com um valor menor. Suponha duas STAs **A** e **B**. **A** gera o valor 0 para *slotCnt*, enquanto **B** gera 1. Ambas tiveram seus relógios reinicializados ao saírem da *location Waiting_DIFS*. Segundo o padrão, a STA **A** deve transmitir primeiro (equação 6.1):

$$\text{backoff} = random() \times aSlotTime \tag{6.1}$$

E substituindo os valores de **A** e **B** na fórmula tem-se a equação 6.2 e equação 6.3, respectivamente:

$$\text{backoff}_A = 0 \times 9 = 0\,\mu\text{s} \tag{6.2}$$

$$\text{backoff}_B = 1 \times 9 = 9\,\mu\text{s} \tag{6.3}$$

Constata-se que **A** não espera nenhum tempo a mais, já que esperou por DIFS. **B** ainda esperará um tempo a mais, 9 µs. Contudo, no modelo, é possível **B** ter prioridade sobre **A**, isto é, **B** transmite antes do que **A**, mesmo **B** esperando 9 µs. A invariante *x <= TslotMicroSecs* especifica o tempo máximo permitido no estado, não o tempo mínimo de permanência. Dessa forma, **A** pode sair de *Waiting_Slot* em um valor de tempo entre 0 e 9 µs, inclusive. Somente quando *x == 9* **A** estará obrigada a sair de *Waiting_Slot* e quando *x == 9*, **B** pode decrementar *slotCnt* e tomar a transição antes de **A**, o que é semanticamente incorreto. Repare que **A** e **B** chegam em *Waiting_Slot* com *x = 0* em razão de *x* ser sempre reiniciado após a *location Waiting_DIFS* e a *location* para gerar o número aleatório ser *committed* (o tempo não passa). Não se conseguiu resolver este problema.

### 6.9.3 Números Aleatórios

Outro ponto deparado é que a geração de números aleatórios não está rigorosamente dentro do especificado em IEEE Computer Society, LAN/MAN Standards Committee e Institute of Electrical and Electronics Engineers (2007, p.260) que descreve que o número aleatório é escolhido de uma distribuição uniforme (todos os números com a mesma probabilidade de ocorrência). Tanto no modelo elaborado por Rosas (2014) como no elaborado neste trabalho, a distribuição não é uniforme. O valor de um número $n$ dentro do intervalo $[0, CW]$ é dado pela equação 6.4.

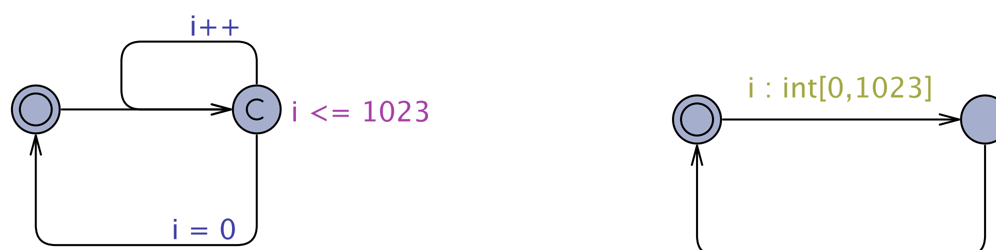
$$\Pr(n) = \begin{cases} \dfrac{1}{2} & n == 0 \\ \left(\dfrac{1}{2}\right)^n & n \neq 0 \end{cases} \tag{6.4}$$

Uma solução concebida é usar a *feature select* (Seção 6.1) para gerar os números aleatórios. Com intenção de verificar que isso não incorreria em um aumento de complexidade, isolou-se o mecanismo de geração de números aleatórios do modelo original comparando-o com um outro mecanismo que usa o *select* para a geração de números aleatórios. Para descobrir e comparar o número de estados e transições possíveis de cada solução, usou-se o *model checker* DIVINE verificando a propriedade de *deadlock* – percorre todos os estados e transições possíveis.[6] Para o modelo original foram percorridos 1025 estados e 2048 transições e para o

[6] Comando utilizado: `divine verify`.

modelo com *select* foram percorridos 2 estados e 1025 transições. Reduziu-se, assim, de modo significativo a quantidade de estados e transições. A Figura 30a ilustra o modelo original enquanto a Figura 30b ilustra o modelo usando *select*. O valor 1023 nas imagens refere-se ao valor máximo possível assumido pela janela de contenção $CW$ dado por $aCWmax$ (IEEE Computer Society; LAN/MAN Standards Committee; Institute of Electrical and Electronics Engineers, 2007).

Contudo, não foi possível implementar o gerador de números aleatórios proposto. A razão é que os valores utilizados na *feature select* devem ser calculados em tempo de compilação e o valor máximo da janela de contenção (*cw*) é variável, ou seja, não é computado em tempo de compilação. Estabelecer um valor fixo faria com que o modelo estivesse fora do padrão IEEE 802.11. Assim, optou-se por fazer uso do mecanismo como na Figura 30a.




**(a) Gerador de números aleatórios usado em Rosas (2014)**

**(b) Gerador de números aleatórios proposto**

**Figura 30 – Geradores de números aleatórios**

**Fonte: Autoria própria**

### 6.9.4 Variável atribuída com valor *out of range*

Durante a verificação de *deadlock* ocorreu uma atribuição à variável *numOfTx* com o valor 32.728, maior que o máximo permitido em variáveis *int*: 32.727.

A solução neste caso também foi bem simples: basta remover o comando *numOfTx*++ do modelo. Como a variável é utilizada somente para controle e não tem nenhum *guard* relacionado não há maiores problemas em removê-la.

### 6.9.5 Informações Técnicas

Um aspecto do trabalho de Rosas (2014) que demonstrou escassez de informações é detalhamento do ambiente de verificação. A configuração de *hardware* (CPU, RAM, etc.) da

máquina onde a verificação foi realizada não foi detalhada. Faltaram ainda informações sobre os resultados da verificação: tempo de execução, memória utilizada, a quantidade de estados percorridos (fornecida pela versão *cli* do UPPAAL) e parâmetros utilizados na versão de linha de comando do UPPAAL (*verifyta*). Esse último é facilmente deduzido, já que, para garantir ausência de *deadlock*, somente as opções *S*<*n*> e *A* podem ser consideradas pois as outras opções sacrificam confiabilidade pela performance.

# 7 RESULTADOS

O presente capítulo descreve os resultados obtidos. Primeiro, a Seção 7.1 apresenta a configuração dos ambientes de verificação. Em seguida a Seção 7.3 lista os parâmetros de configuração utilizados no UPPAAL. A Seção 7.4 discute os resultados obtidos e finalizando a Seção 7.5 contém uma tabela resumindo os resultados obtidos.

## 7.1 DETALHES DO AMBIENTE DE VERIFICAÇÃO

A verificação de propriedades ocorreu em uma máquina com configuração listada no Quadro 3. Tentou-se verificar se a propriedade de *deadlock* é ou não satisfeita em um servidor da Universidade Federal do Paraná (UFPR)[1] mas não foi possível devido a explosão de estados.

| **Configuração** | |
|---|---|
| **Sistema Operacional** | Ubuntu 16.04 LTS |
| **Memória** | 6 GB |
| **CPU** | Intel Core i3-3110M 2.4 GHz |
| **Disco** | 500 GB |
| **Versão do UPPAAL** | 4.1.19, 64 bits |

**Quadro 3 – Configurações dos ambientes de verificação**

**Fonte: Autoria própria**

## 7.2 GERAÇÃO DOS INVERVALOS DE TEMPO

Para gerar os intervalos de tempo utilizados na verificação utilizou-se um *script bash* que internamente faz uso do GNU Octave para calcular os intervalos de acordo com os parâmetros especificados. Os parâmetros utilizados foram:

- *channel width* de 10 MHz;
- *data rate* de 12 Mbps;
- *control rate* de 12 Mbps;
- *payload size* de 1000 *bytes*.

[1] Os processos do UPPAAL eram limitados ao uso de no máximo 20 GB; passado esse limite os processos eram terminados.

## 7.3 PARÂMETROS DE VERIFICAÇÃO DO UPPAAL

A versão *cli* do UPPAAL utilizada neste trabalho, *verifyta*, possui diferentes parâmetros de configuração para o processo de verificação. O parâmetro configurado neste trabalho foi a redução do espaço de estados (*state space reduction*). Há outros parâmetros disponíveis para utilização, mas eles não garantem que todos os estados de uma propriedade especificada sejam verificados. Esses parâmetros geralmente fazem com que o processo de verificação seja mais rápido. Tem-se, então, uma margem para a possibilidade de uma propriedade ser satisfeita ou não. O parâmetro é especificado através da opção *-S<number>* onde *<number>* varia de 0 a 2 (BEHRMANN; DAVID; LARSEN, 2006):

- **0** indica para não usar nenhuma técnica de compressão no espaço de estados gerados, o que acelera a verificação mas consome muita memória;
- **1** indica o uso de técnicas conservativas para compressão balanceando entre quantidade de memória utilizada e velocidade e;
- **2** indica para comprimir ainda mais os estados gerados mas aumentando o tempo de verificação. A opção utilizada foi *-S2*.

## 7.4 RESULTADOS

Esta seção apresenta os resultados obtidos da verificação. Cada subseção exibe se a fórmula foi satisfeita ou não, a quantidade de estados armazenados e explorados, o tempo de CPU utilizado e quantidade de memória virtual e residente utilizada pelo UPPAAL. Quando referido que não foi possível realizar a verificação de uma propriedade, isso deve-se ao problema da explosão de estados que requer uma quantidade de memória muito alta para armazenamento dos estados.

Há de se acresecentar ainda que a propriedade de *deadlock* não foi passível de verificação devido a explosão de estados sendo verificada no modelo simplificado (Seção 6.8). Como a verificação se deu na versão com interface gráfica, informações técnicas (uso de memória, tempo de uso da CPU, etc.) ao término da verificação não são exibidas e por isso não foram aqui listadas.

### 7.4.1 *Deadlock*

A primeira propriedade (7.1) é do tipo de segurança e é referente a ausência de *deadlock* no modelo. Não foi possível verificar essa propriedade. Para o modelo simplificado foi possível

verificá-la para *n* = *2*. A Figura 31 exibe os resultados da verificação para o modelo simplificado.

$$A[]\ not\ deadlock \tag{7.1}$$

```
Verifying formula 1 at line 4
 -- Formula is satisfied.
 -- States stored : 28872255 states
 -- States explored : 28896782 states
 -- CPU user time used : 266140 ms
 -- Virtual memory used : 3053476 KiB
 -- Resident memory used : 3009020 KiB
```

**Figura 31 – Verificação da propriedade de *deadlock***

**Fonte: Autoria própria**

A verificação foi relativamente rápida, cerca de quatro minutos e meio. Contudo, aumentando o valor de *n* para *3*, tem-se uma explosão de estados fazendo com que o tempo de verificação aumente consideravelmente.

### 7.4.2 *Nenhuma transmissão em* $T_n$

A propriedade especificada pela 7.2 especifica que é possível (*E<> exists*) que uma dada STA (*i : int[0, N-1]*) não transmita (*Node(i).numOfTx == 0*) em um intervalo de tempo definido ($T_n$). A propriedade foi verificada e satisfeita para $n = 3$. A Figura 32 apresenta os resultados.

$$E <> exists(i : int[0, N-1])(y \geq T_n \ \&\& \ Node(i).numOfTx == 0) \tag{7.2}$$

```
Verifying formula 1 at line 3
 -- Formula is satisfied.
 -- States stored : 431672 states
 -- States explored : 819838 states
 -- CPU user time used : 4400 ms
 -- Virtual memory used : 91616 KiB
 -- Resident memory used : 62696 KiB
```

**Figura 32 – Verificação da propriedade nenhuma transmissão em $T_n$**

**Fonte: Autoria própria**

### 7.4.3 *k* transmissões em $T_n$

A seguinte propriedade de vivacidade (7.3) especifica que é possível que uma certa STA transmita *k* vezes ($= 2$, $Node(i).numOfTx \geq 2$ && $Node(i).numOfTx \leq N$) em um intervalo de tempo definido ($y \ \leq T_n$). A propriedade foi verificada para valores de *n* até 3 sendo o resultado da verificação listado na Figura 33.

$$E <> exists(i : int[0, N-1])(y \leq T_n \ \&\& \ Node(i).numOfTx \geq 2 \ \&\& \ Node(i).numOfTx \leq N) \quad (7.3)$$

```
Verifying formula 1 at line 3
 -- Formula is satisfied.
 -- States stored : 8576 states
 -- States explored : 12347 states
 -- CPU user time used : 70 ms
 -- Virtual memory used : 42744 KiB
 -- Resident memory used : 7892 KiB
```

**Figura 33 – Verificação da propriedade *k* transmissões em $T_n$**

**Fonte: Autoria própria**

### 7.4.4 *Transmissões simultâneas*

Esta propriedade (7.4) de vivacidade especifica que é possível ocorrer uma transmissão simultânea no modelo, o que é indicado por uma STA em *Transmitting_Data* e outra STA diferente de *i* em *Transmitting_Data_STX*. Verificada para *n = 2* já que apenas duas transmissões podem ocorrer simultaneamente. Na Figura 34 tem-se os resultados.

$$E <> exists(i : int[0, N-1])(Node(i).Transmitting_Data_PTX \ \&\& \ Node(not\ i).Transmitting_Data_STX) \quad (7.4)$$

```
Verifying formula 1 at line 3
 -- Formula is satisfied.
 -- States stored : 3 states
 -- States explored : 11 states
 -- CPU user time used : 0 ms
 -- Virtual memory used : 41304 KiB
 -- Resident memory used : 6112 KiB
```

**Figura 34 – Verificação da propriedade de transmissões simultâneas**

**Fonte: Autoria própria**

#### 7.4.5 Transmissões simultâneas terminam ao mesmo tempo

A propriedade de vivacidade (7.5) especifica que tanto a transmissão primária como a transmissão secundária terminam ao mesmo tempo visando garantir o funcionamento correto do modelo. Verificada para *n = 2* já que somente duas transmissões podem ocorrer ao mesmo tempo. A propriedade pode ser subdividida em duas partes. A primeira, baseada na propriedade anterior, $(Node(i).Transmitting_Data_PTX \ \&\& \ Node(not\ i).Transmitting_Data_STX)$ especifica que é possível ocorrer uma transmissão simultânea. A segunda parte, $(Node(i).x == TonlyData \ \&\& \ Node(not\ i).x == TonlyData)$, especifica que os relógios de ambas STAs tem o valor *TonlyData* que representa somente os dados de uma transmissão. Assim, percebe-se que as transmissões terminam simultaneamente. Os resultados são apresentados na Figura 35.

$$
\begin{aligned}
E <> exists(i : int[0, N-1]) \\
(Node(i).Transmitting_Data_PTX \ \&\& \\
Node(not\ i).Transmitting_Data_STX) \ \&\& \\
(Node(i).x == TonlyData \ \&\& \ Node(not\ i).x == TonlyData)
\end{aligned} \tag{7.5}
$$

```
Verifying formula 1 at line 3
 -- Formula is satisfied.
 -- States stored : 3 states
 -- States explored : 11 states
 -- CPU user time used : 0 ms
 -- Virtual memory used : 41328 KiB
 -- Resident memory used : 6152 KiB
```

**Figura 35 – Verificação da propriedade de transmissões simultâneas terminam simultaneamente**

**Fonte: Autoria própria**

Como pretendeu-se verificar a funcionalidade de uma transmissão *full duplex*, a verificação foi rápida.

### 7.4.6 ACK na transmissão primária

A propriedade de vivacidade listada em 7.6 formaliza que, para uma transmissão primária concluída com sucesso ($Node(i).Read_Destiny$ && $Node(i).dst == Node(i).id$), ocorre um ACK $Node(i).x == Tack$. A propriedade foi verificada para $n = 2$ uma vez que o foco é na transmissão *full-duplex*. O resultado da verificação pode ser visto na Figura 36.

$$E <> exists(i : int[0, N-1])(Node(i).Read_Destiny \ \&\& \ Node(i).dst == Node(i).id \ \&\& \ Node(i).x == Tack) \quad (7.6)$$

```
Verifying formula 1 at line 3
 -- Formula is satisfied.
 -- States stored : 5 states
 -- States explored : 21 states
 -- CPU user time used : 0 ms
 -- Virtual memory used : 41316 KiB
 -- Resident memory used : 6124 KiB
```

**Figura 36 – Verificação da propriedade de ACK na transmissão primária**

**Fonte: Autoria própria**

A verificação não demorou muito tempo por não haver muitas STAs e também por ter como foco o término correto de uma transmissão *full duplex*.

### 7.4.7 ACK na transmissão secundária

A propriedade de vivacidade listada em 7.7 especifica que, para uma transmissão secundária concluída corretamente, ocorre um ACK secundário. Verificou-se a propriedade para $n = 2$ uma (foco na transmissão *full-duplex*). A Figura 37 apresenta o resultado da verificação.

$$E <> exists(i : int[0, N-1])(Node(i).Read_ACK_STX \ \&\& \ Node(i).dst == Node(i).id) \quad (7.7)$$

Novamente a verificação foi rápida devido ao objetivo de verificar a propriedade somente para uma transmissão *full duplex*.

```
Verifying formula 1 at line 3
 -- Formula is satisfied.
 -- States stored : 6 states
 -- States explored : 23 states
 -- CPU user time used : 0 ms
 -- Virtual memory used : 41308 KiB
 -- Resident memory used : 6108 KiB
```

**Figura 37 – Verificação da propriedade de ACK na transmissão secundária**

**Fonte: Autoria própria**

#### 7.4.8 ACK simultâneo

A próxima propriedade de vivacidade (7.8) especifica que ocorre um ACK simultâneo em um transmissão *full-duplex*. Verificou-se para *n = 2*.

$$E <> exists(i : int[0, N-1])(ack_success == true \,\&\& \\ Node(i).Read_ACK_STX \,\&\&\, Node(i).dst == Node(i).id) \quad (7.8)$$

```
Verifying formula 1 at line 3
 -- Formula is satisfied.
 -- States stored : 6 states
 -- States explored : 23 states
 -- CPU user time used : 0 ms
 -- Virtual memory used : 41308 KiB
 -- Resident memory used : 6076 KiB
```

**Figura 38 – Verificação da propriedade de ACK simultâneo**

**Fonte: Autoria própria**

Como as duas propriedades anteriores especificam que ocorre tanto um ACK para a primeira transmissão como para a segunda transmissão, um ACK simultâneo também ocorre.

### 7.5 VISÃO GERAL DOS RESULTADOS

O Quadro 4 resume os resultados obtidos neste trabalho.[2]

As propriedades de vivacidade são mais rápidas de serem verificadas já que basta encontrar um caminho que satisfaça a propriedade, o que não é válido para as propriedades de segurança, para as quais não pode ser encontrado nenhum estado satisfazendo a propriedade (ou dentro de um intervalo de tempo como no caso da propriedade de nenhuma transmissão em $T_n$).

[2] A tabela é para o modelo completo e não para o modelo simplificado.

| Propriedade | n | Tipo | Verificada |
|---|---|---|---|
| *Deadlock* | 2 | Segurança | Não |
| Nenhuma transmissão em $T_n$ | 3 | | Sim |
| k transmissões em $T_n$ | 3 | Vivacidade | Sim |
| Transmissões simultâneas | 2 | | Sim |
| Transmissões simultâneas terminam ao mesmo tempo | 2 | | Sim |
| ACK na transmissão primária | 2 | | Sim |
| ACK na transmissão secundária | 2 | | Sim |
| ACK simultâneo | 2 | | Sim |

**Quadro 4 – Resultados**

**Fonte: Autoria própria**

Os valores de *n* escolhidos foram baixos. O principal fator que contribuiu para isso foi a explosão de estados incorrendo em uma demora no tempo de verificação. Portano, para aumentar os valores utilizados, é necessário refinar o modelo preservando as funcionalidades esperadas possivelmente identificando alguma parte que esteja contribuindo para a explosão de estados.

# 8 CONCLUSÃO

Nos capítulos anteriores foram apresentados diversos conceitos pertinentes à realização deste trabalho. *Model checking* foi descrito juntamente com os formalismos para a modelagem de sistemas e a especificação de propriedades (Capítulo 2). Logo após os *model checkers* SPIN, PRISM, UPPAAL e UPPAAL SMC (Capítulo 3) foram abordados. Cada um foi exposto, de modo breve e, após uma análise (Seção 3.6), optou-se pela utilização do UPPAAL. A utilização de outras ferramentas para o (factível) seguimento deste trabalho fica em aberto, sendo o PRISM de maior interesse dada a considerável e crescente quantidade de trabalhos publicados que fazem uso dele na literatura. Descreveu-se, posteriormente, o funcionamento do protocolo CSMA/CA adotado no padrão IEEE 802.11 e o modo de transmissão RTS/CTS, utilizado para prevenir o problema do terminal escondido (Capítulo 4). Viu-se ainda um protocolo de rede *full duplex* para redes sem fio recentemente desenvolvido, algo que não era possível até alguns anos atrás, o que pode possibilitar relevantes avanços para a área de redes. Trabalhos relacionados contextualizaram como o *model checking* tem sido aplicado em trabalhos de verificação formal (Capítulo 5). Seguiu-se o desenvolvimento de um modelo para o protocolo de rede sem fio *full duplex* adotado (Capítulo 6). Por último, os resultados da verificação das propriedades foram então apresentados (Capítulo 7).

A princípio foi proposto, como objetivo geral deste trabalho, a verificação formal do protocolo *Piece-by-Piece* (PbP) proposto por Queiroz e Hexsel (2015). Sem embargo, o objetivo foi alterado para a verificação de um protocolo de rede *full duplex*. Isso decorreu primariamente pelo motivo de que protocolos *full duplex* são originais e não há, até onde sabe-se, nenhum trabalho relacionado na área sobre a verificação dos mesmos.

Como apresentado na Seção 3.6, para a realização do presente trabalho optou-se pelo UPPAAL pelas seguintes razões: (*i*) em Rosas (2014) foi usado o UPPAAL para verificar formalmente o CSMA/CA do padrão 802.11; (*ii*) possui modelagem gráfica facilitando assim a construção do modelo; (*iii*) por ser adequado para protocolos de comunicação de redes; (*iv*) e por permitir a criação de modelos através de uma interface gráfica o que é desejável para a elaboração deste trabalho.

As principais contribuições deste trabalho são:

- ☒ Publicação de um suscinto artigo abrangendo o referencial a respeito de *model checking* elaborado na primeira parte deste trabalho (JUNIOR; ALVES, 2015);
- ☒ Análise e refinamento do modelo desenvolvido em Rosas (2014);
- ☒ Elaboração formal de um modelo completo do protocolo escolhido com uma versão simplificada na qual foi possível verificar a ausência de *deadlock*;
- ☒ Especificação e verificação de propriedades formais referentes ao protocolo escolhido.

O principal obstáculo na realização deste trabalho foi o problema da explosão de estados que implica em: grande quantidade de memória utilizada e demora na verificação. A quantidade de memória consumida é extremamente alta considerando a razoável complexidade do modelo, chegando a consumir mais de 200 GB de RAM. Pela quantidade de estados ser alta, acaba-se fazendo uso do *swap*, o que faz com que a verificação demore muito mais já que discos rígidos são consideravelmente mais lentos que a memória RAM, principalmente se for realizada em uma máquina com discos rotacionais em vez de discos de unidade de estado sólido (*solid state drive*).

Trabalhos futuros incluem:

- pesquisa sobre utilização de *model checkers multicore* como o DIVINE;
- estudo e aplicação de *model checking* probabilístico;
- definição de propriedades adicionais;
- verificação formal do protocolo PbP.

# REFERÊNCIAS


ADMINISTRATION, F. A. Governmental, **Federal Register Airworthiness Directives - The Boeing Company Airplanes**. 2015. Disponível em: <https://www.federalregister.gov/articles/2015/05/01/2015-10066/airworthiness-directives-the-boeing-company-airplanes>.

ALUR, R. Timed automata. In: **Computer Aided Verification**. Springer, 1999. p. 8–22. Disponível em: <http://link.springer.com/chapter/10.1007/3-540-48683-6_3>.

ALUR, R.; DILL, D. L. A Theory of Timed Automata. **Theoretical Computer Science**, v. 126, p. 182–225, 1994.

BABICH, F.; DEOTTO, L. Formal methods for specification and analysis of communication protocols. **Communications Surveys & Tutorials, IEEE**, v. 4, n. 1, p. 2–20, 2002. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5341329>.

BAIER, C.; KATOEN, J.-P. **Principles of model checking**. Cambridge, Mass: The MIT Press, 2008. ISBN 978-0-262-02649-9 0-262-02649-X.

BARBOZA, F. J. *et al.* Specification and Verification of the IEEE 802.11 Medium Access Control and an Analysis of its Applicability to Real-Time Systems. **Electronic Notes in Theoretical Computer Science**, v. 195, p. 3–20, jan. 2008. ISSN 15710661. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1571066108000091>.

BARNAT, J. *et al.* DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: **Computer Aided Verification (CAV 2013)**. [S.l.]: Springer, 2013. (LNCS, v. 8044), p. 863–868.

BEHRMANN, G.; DAVID, A.; LARSEN, K. G. A tutorial on uppaal. In: **Formal methods for the design of real-time systems**. Springer, 2006. p. 200–236. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-540-30080-9_7>.

BEN-OTHMAN, J.; MOKDAD, L.; BOUAM, S. AMCLM: ADAPTIVE MULTI-SERVICES CROSS-LAYER MAC PROTOCOL FOR IEEE 802.11 NETWORKS. **Journal of Interconnection Networks**, v. 10, n. 04, p. 283–301, dez. 2009. ISSN 0219-2659, 1793-6713. Disponível em: <http://www.worldscientific.com/doi/abs/10.1142/S0219265909002583>.

BENTLEY, B. Validating the Intel (R) Pentium (R) 4 microprocessor. In: **Design Automation Conference, 2001. Proceedings**. IEEE, 2001. p. 244–248. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=935512>.

BOCHMANN, G. V. Finite state description of communication protocols. **Computer Networks (1976)**, v. 2, n. 4-5, p. 361–372, set. 1978. ISSN 03765075. Disponível em: <http://dx.doi.org/10.1016/0376-5075(78)90015-6>.

DAVID, A. *et al.* Uppaal smc tutorial. **International Journal on Software Tools for Technology Transfer**, p. 1–19, 2015. Disponível em: <http://link.springer.com/article/10.1007/s10009-014-0361-y>.

FRUTH, M. Probabilistic model checking of contention resolution in the IEEE 802.15. 4 low-rate wireless personal area network protocol. In: **Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on**.

IEEE, 2006. p. 290–297. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4463726>.

HAMMAL, Y. *et al.* Formal modeling and verification of an enhanced variant of the IEEE 802.11 CSMA/CA protocol. **Journal of Communications and Networks**, v. 16, n. 4, p. 385–396, ago. 2014. ISSN 1229-2370. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6896562>.

HAVELUND, K.; LOWRY, M.; PENIX, J. Formal analysis of a space-craft controller using SPIN. **IEEE Transactions on Software Engineering**, v. 27, n. 8, p. 749–765, ago. 2001. ISSN 00985589. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=940728>.

HEUSSE, M. *et al.* Performance anomaly of 802.11b. In: . IEEE, 2003. v. 2, p. 836–843. ISBN 978-0-7803-7752-3. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1208921>.

HOLZMANN, G. J. The model checker SPIN. **IEEE Transactions on software engineering**, n. 5, p. 279–295, 1997. Disponível em: <http://www.computer.org/csdl/trans/ts/1997/05/e0279.pdf>.

IEEE Computer Society; LAN/MAN Standards Committee; Institute of Electrical and Electronics Engineers. **IEEE standard for information technology part 11, wireless LAN medium access control (MAC) and physical layer (PHY) specifications part 11, wireless LAN medium access control (MAC) and physical layer (PHY) specifications**. New York, N.Y.: Institute of Electrical and Electronics Engineers, 2007. ISBN 978-0-7381-5655-2 978-0-7381-5656-9. Disponível em: <http://ieeexplore.ieee.org/servlet/opac?punumber=4248376>.

JAIN, M. *et al.* Practical, real-time, full duplex wireless. In: **Proceedings of the 17th annual international conference on Mobile computing and networking**. ACM, 2011. p. 301–312. Disponível em: <http://dl.acm.org/citation.cfm?id=2030647>.

JENSEN, H. E.; LARSEN, K. G.; SKOU, A. Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. **BRICS Report Series**, v. 3, n. 24, 1996. Disponível em: <http://ojs.statsbiblioteket.dk/index.php/brics/article/view/20005>.

JUNIOR, M. P.; ALVES, G. V. A Study Towards the Application of UPPAAL Model Checker. In: **WEIT - III Workshop - Escola de Informática Teórica**. [S.l.]: UFRGS, 2015. p. 19–26.

KWIATKOWSKA, M.; NORMAN, G.; PARKER, D. PRISM 4.0: Verification of probabilistic real-time systems. In: **Computer aided verification**. Springer, 2011. p. 585–591. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-22110-1_47>.

KWIATKOWSKA, M.; NORMAN, G.; SPROSTON, J. **Probabilistic model checking of the IEEE 802.11 wireless local area network protocol**. Springer, 2002. Disponível em: <http://link.springer.com/chapter/10.1007/3-540-45605-8_11>.

LAMPORT, L. What Good is Temporal Logic? **Information Processing**, v. 83, p. 657–668, 1983.

LIMA, V. *et al.* Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages. **Electronic Notes in Theoretical Computer Science**, v. 254, p. 143–160, out. 2009. ISSN 15710661. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1571066109004186>.

MATEO, J. A. *et al.* Probabilistic Model Checking: One Step Forward in Wireless Sensor Networks Simulation. **International Journal of Distributed Sensor Networks**, v. 2015, p. 1–11, 2015. ISSN 1550-1329, 1550-1477. Disponível em: <http://www.hindawi.com/journals/ijdsn/2015/285396/>.

OXFORD, U. o. **PRISM Manual | Main / Welcome**. 2015. Disponível em: <http://www.prismmodelchecker.org/manual/Main/Welcome>.

QADIR, J.; HASAN, O. Applying Formal Methods to Networking: Theory, Techniques, and Applications. **Communications Surveys & Tutorials, IEEE**, v. 17, n. 1, p. 256–291, 2015. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6873212>.

QUEIROZ, S.; HEXSEL, R. Translating Full Duplexity into Capacity Gains for the High-Priority Traffic Classes of IEEE 802.11. 2015. Disponível em: <http://www.researchgate.net/profile/Saulo_Queiroz/publication/269634844_Translating_Full_Duplexity_into_Capacity_Gains_for_the_High-Priority_Traffic_Classes_of_IEEE_802.11/links/551c09300cf20d5fbde24c2f.pdf>.

ROSAS, F. A. Verificação Formal de um Protocolo de Rede sem Fio Através de Model Cheking. **Trabalho de Conclusão de Curso**, Ponta Grossa, 2014.

STOLLER, S. D. Model-Checking Multi-threaded Distributed Java Programs. In: GOOS, G. *et al.* (Ed.). **SPIN Model Checking and Software Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. v. 1885, p. 224–244. ISBN 978-3-540-41030-0 978-3-540-45297-3. Disponível em: <http://link.springer.com/10.1007/10722468_14>.

VAANDRAGER, F. A First Introduction to uppaal. **Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook**, v. 18, 2011. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.457.5080&rep=rep1&type=pdf#page=20>.

YI, W.; PETTERSSON, P.; DANIELS, M. Automatic verification of real-time communicating systems by constraint-solving. In: **FORTE**. Citeseer, 1994. v. 6, p. 243–258. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.474.6706&rep=rep1&type=pdf>.

# APÊNDICE A – DOCUMENTAÇÃO DO MODELO ELABORADO

Este apêndice documenta todas as *locations* e transições do modelo *full duplex* visto na Figura 28 da Seção 6.7.

A nomeação das *locations* segue o seguinte critério: *locations urgent* ou *commited* não possuem nome terminando em *-ing*. Dessa forma o nome fica consistente com a semântica da *location*. As demais possuem *-ing* em seus nomes para identificar um estado contínuo temporário.

1. **Generate_RN** : *location* responsável por gerar um número aleatório entre $0$ e $cw$. O *guard* $cnt \geq 0$ é responsável pelo limite inferior enquanto a invariante $cnt \leq cw$ é responsável pelo limite superior. É marcada como *committed* para que o tempo não passe e haja prioridade para a geração do número aleatório, *interleaving* só acontece com outras *locations committed*.

2. **Read_ACK_STX** : *location* que modela a leitura do ACK pela STA receptora. Supõe que a leitura do ACK é instantânea já que é marcada como urgente.

3. **Read_Destiny** : *location* que representa a leitura de um cabeçalho na transmissão primária. Supõe leitura instantânea do ACK.

4. **Read_Header_PTX** : modela a leitura de um cabeçalho da transmissão primária. Marcada como *commited* para que haja prioridade na leitura do cabeçalho.

5. **Replying_Header_PTX** : uma STA está respondendo o cabeçalho enviado pela STA *primary sender*. O tempo de envio é dado pela invariante $x \leq (TPrimary_Timer - 1)$.

6. **Start** : *location* inicial da qual todas as STAs começam. Marcada como urgente para que todas as STAs executem suas funções de inicialização esperando por DIFS logo em seguida.

7. **Sending_ACK_PTX** : modela o envio do ACK da transmissão primária, o que leva $TACK$, invariante $x \leq TACK$.

8. **Sensing_for_DIFS** : *location* que modela a espera inicial para início de uma transmissão. Uma transição será executada após DIFS unidades de tempo, invariante $x \leq DIFS$.

9. **Transmitting_Data_PTX** : modela a transmissão primária realizada pela STA *primary sender*. O tempo da transmissão é $TPayload$.

10. **Transmitting_Data_STX** : modela a transmissão secundária realizada pela STA *primary receiver*. O tempo da transmissão é $TPayload$.

11. **Transmitting_Header_PTX** : modela o envio do cabeçalho pela STA *primary sender*. O tempo de envio é dado por $THeader$.

12. **Waiting_ACK_PTX** : *location* que representa uma STA *primary sender* esperando pelo recebimento de um ACK. O tempo máximo de espera pelo ACK é de $ACKTimeout$.

13. **Waiting_ACK_STX** : *location* que representa uma STA *primary receiver* esperando pelo recebimento de um ACK. O tempo máximo de espera pelo ACK é de $ACKTimeout$.

14. **Waiting_DIFS** : essa *location* marca a espera de DIFS ininterrupto, $x \leq DIFS$. Se houver uma transmissão durante a espera, a STA muda para *Waiting_Medium_Free*.

15. **Waiting_Medium_Free** : *location* que modela o meio como ocupado. As STAs devem esperar o mesmo a ser liberado.

16. **Waiting_Response** : após enviar o cabeçalho, a STA *primary sender* espera por no máximo $TPrimary_Timer$ por uma resposta da STA *primary receiver*.

17. **Waiting_SIFS_STX** : modela a espera por SIFS antes do envio do ACK para a STA *primary receiver*.

18. **Waiting_Slot** : modela uma STA esperando pelo tempo de *backoff*.

As transições "ativas", aquelas marcadas com *!*, fazem uso da voz ativa no verbo (*synchronize*), enquando as "passivas", seguidas por *?*, utilizam voz passiva (*synchronized*). Além disso, em vez de usar a ordem dos elementos de transição do UPPAAL *select*, *guard*, *sync* e *update*, a ordem utilizada nesta documentação é *guard*, *select*, *sync* e *update*.

1. **Generate_RN** para

   - **Generate_RN** : transição não determinística que incrementa o valor do número aleatório em uma unidade.
     - update $cnt++$
   - **Waiting_Slot** : transição não determinística que encerra a geração do número aleatório atualizando as devidas variáveis.
     - when $cnt \geq 0$
     - update $slotCnt = cnt, cnt = 0$

2. **Read_ACK_STX** para

   - **Waiting_DIFS** : ACK recebido com sucesso dentro do tempo esperado. Atualize as variáveis necessárias e indique uma transmissão realizada com sucesso.
     - when $dst == id$
     - update $x = 0, cw = aCWmin, nTx = 0, numOfTx++$
   - **Waiting_DIFS** : ACK não foi destinado à esta STA.
     - when $x \neq id$

- update $x = 0$

3. **Read_Destiny** para

- **Waiting_DIFS** : ACK recebido dentro do tempo esperado e destinada à STA correta. Atualize as variáveis, indique uma transmissão ocorrida com sucesso e sinalize o término do envio do ACK secundário.
  - when $dst == id$ and $x == Tack$
  - synchronize with other STAs on channel $send_stx_data!$ and
  - update $x = 0, cw = aCWmin, numOfTx + +$
- **Waiting_DIFS** : ACK destinado à STA errada.
  - when $dst \neq id$
  - update $x = 0$

4. **Read_Header_PTX** para

- **Replying_Header_PTX** : o *id* da STA corresponde ao destino do cabeçalho.
  - when $dst == id$
  - update $x = 0$
- **Waiting_Medium_Free** : o *id* da STA é diferente do destino do cabeçalho.
  - when $dst \neq id$

5. **Replying_Header_PTX** para

- **Transmitting_Data_STX** : após o término do envio da resposta ao cabeçalho, estabelece-se uma transmissão *full duplex* somente se houver uma transmissão em andamento.
  - when $x == (TPrimary_Timer - 1)$ and $nTx == 1$
  - synchronize with other STAs on channel $reply_header!$ and
  - update $x = Tlatency$
- **Waiting_DIFS** : mais do que uma STA transmitiu. Colisão ocorre.
  - when $nTx \neq 1$
  - update $x = 0$

6. **Start** para

- **Sensing_for_DIFS** : uma STA executa sua função de inicialização.
  - when $x == Tack$
  - synchronize with other STAs on channel $finish_sending!$ and
  - update $x = 0$

7. **Sending_ACK_PTX** para
    - **Waiting_ACK_STX** : uma STA termina o envio do ACK para a transmissão primária.
        - when $x == Tack$
        - synchronize with other STAs on channel $finish_sending!$ and
        - update $x = 0$
8. **Sensing_for_DIFS** para
    - **Transmitting_Header_PTX** : uma STA de forma não determinística decide iniciar a primeira transmissão no modelo.
        - when $x == DIFS$
        - synchronize with other STAs on channel $start_send!$ and
        - update $x = 0, nTx++$
    - **Waiting_Medium_Free** : uma STA já iniciou uma transmissão, então espera o meio ficar livre novamente.
        - synchronized with other STAs on channel $start_send?$ and
        - update $x = 0$
9. **Transmitting_Data_PTX** para
    - **Waiting_ACK_PTX** : uma STA finaliza uma transmissão primária.
        - when $x == TPayload$
        - synchronize with other STAs on channel $finish_sending!$ and
        - update $x = 0$
10. **Transmitting_Data_STX** para
    - **Waiting_SIFS_STX** : STA *primary receiver* termina uma transmissão secundária.
        - when $x == TPayload$
        - update $x = 0$
11. **Transmitting_Header_PTX** para
    - **Waiting_Response** : STA *primary sender* acaba de enviar o cabeçalho para a STA *primary receiver*.
        - when $x == THeader$
        - create variable $d$ of type $int[0, N-2]$ and
        - synchronize with other STAs on channel $send_header!$ and
        - update $dst = d, x = 0$

12. **Waiting_ACK_PTX** para
    - **Waiting_DIFS** : a STA *primary sender* não recebeu o ACK dentro de tempo limite.
      - when $x == ACKTimeout$
      - update $x = 0, nTx --$ and execute $cw = cw_size(cw)$
    - **Waiting_Medium_Free** : a STA *primary sender* recebeu o ACK dentro do tempo limite.
      - synchronized with other STAs on channel $start_send?$ and
      - update $x = 0$
13. **Waiting_ACK_STX** para
    - **Read_ACK_STX** : a STA *primary receiver* recebeu o ACK dentro de tempo limite.
      - create variable $s$ of type $int[0, N-2]$ and $d$ of type $int[0, N-1]$ and
      - synchronized with other STAs on channel $send_stx_data?$ and
      - update $src = s, dst = d$
    - **Waiting_DIFS** : a STA *primary receiver* não recebeu o ACK dentro de tempo limite.
      - when $x == ACKTimeout$
      - update $x = 0$ and execute $cw = cw_size(cw)$
14. **Waiting_DIFS** para
    - **Generate_RN** : uma STA, após passado DIFS, irá sortear um número aleatório.
      - when $x == DIFS\ slotCnt == -1$
      - update $x = 0$
    - **Waiting_Medium_Free** : uma outra STA, *primary sender* ou *receiver*, iniciou uma transmissão tornando o meio ocupado.
      - synchronized with other STAs on channel $start_send?$ and
      - update $x = 0$
    - **Waiting_Slot** : uma STA irá retomar o processo de *backoff* — o número aleatório foi sorteado anteriormente.
      - when $x == DIFS$ and $slotCnt == -1$
      - update $x = 0$
15. **Waiting_Medium_Free** para
    - **Read_Destiny** : o meio foi liberado com o término da transmissão, *primary sender* ou *receiver*, anterior e a STA irá verificar se a mensagem foi endereçada à ela ou não.
      - create variable $s$ of type $int[0, N-2]$ and $d$ of type $int[0, N-1]$ and

- synchronized with other STAs on channel $finish_sending?$ and
- update $src = s, dst = d$

- **Read_Header_PTX** : a transmissão do cabeçalho pela STA *primary sender* foi completada com sucesso.
  - create variable $s$ of type $int[0, N-2]$ and $d$ of type $int[0, N-1]$ and
  - synchronized with other STAs on channel $send_header?$ and
  - update $src = s, dst = d$
- **Waiting_DIFS** : a STA *primary sender* não recebeu nenhuma resposta da STA *primary receiver* dentro do tempo estabelecido por $TPrimary_Timer$. O meio foi liberado.
  - synchronized with other STAs on channel $free_medium?$ and
  - update $x = 0$

16. **Waiting_Response** para

- **Transmitting_Data_PTX** : a STA *primary sender* recebeu uma resposta da STA *primary receiver* dentro do tempo limite.
  - when $x == TPayload$
  - synchronize with other STAs on channel $finish_sending!$ and
  - update $x = 0$
- **Waiting_DIFS** : *timeout* ocorre porque a STA *primary sender* não recebeu nenhuma resposta da STA *primary receiver* dentro do tempo limite esperado.
  - when $x == TPrimary_Timer$
  - synchronize with other STAs on channel $free_medium!$ and
  - update $x = 0, nTx--$ and execute $cw = cw_size(cw)$

17. **Waiting_SIFS_STX** para

- **Sending_ACK_PTX** : após esperar SIFS, a STA *primary receiver* inicia a transmissão do ACK para a STA *primary sender*.
  - when $x == SIFS$
  - synchronize with other STAs on channel $start_send!$ and
  - update $x = 0$

18. **Waiting_Slot** para

- **Transmitting_Header_PTX** : passado o tempo de *backoff*, uma STA inicia uma transmissão primária.
  - when $slotCnt == 0$

- synchronize with other STAs on channel $start_send!$ and
- update $slotCnt = -1, nTx++, x = 0$

- **Waiting_Medium_Free** : uma outra STA já iniciou uma transmissão primária. Esta STA detecta o meio como ocupado e passa a esperar o término da transmissão atual.
  - when $slotCnt \neq 0$
  - synchronized with other STAs on channel $start_send?$ and
  - update $x = 0$

- **Waiting_Slot** : passado o tempo de um SLOT, a STA volta a esperar pelo tempo de mais um SLOT. Repete-se o processo até que o tempo de *backoff* tenha passado.
  - when $x == SLOT$ and $slotCnt \neq 0$
  - update $slotCnt--, x = 0$

# ANEXO A – MODEL CHECKING E O PADRÃO 802.11 DA IEEE, DCF

Este anexo apresenta o modelo elaborado em Rosas (2014).

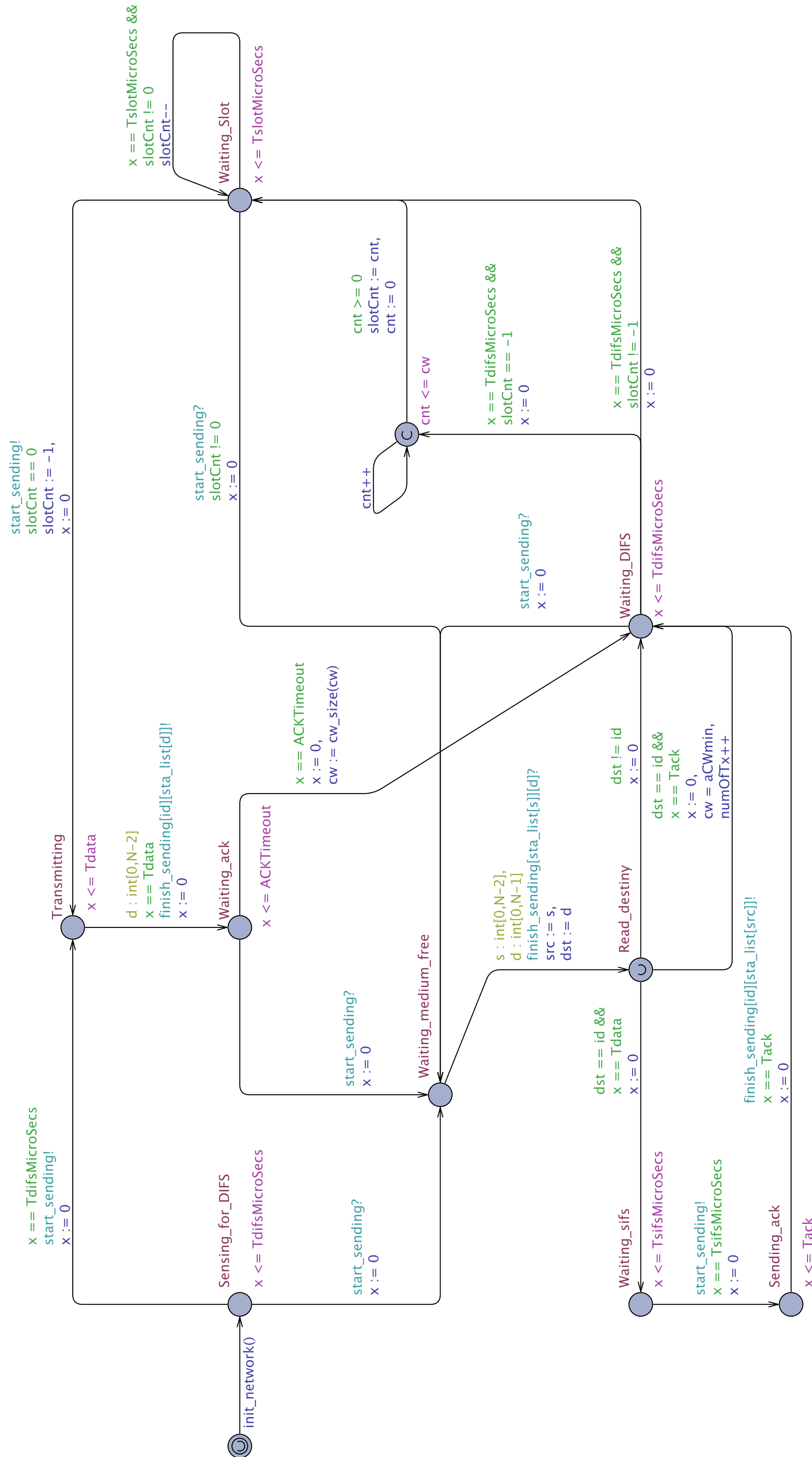



**Figura 39 – Modelo completo**

**Fonte: Rosas (2014, p.49)**