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

FABIANO ALMEIDA ROSAS

VERIFICAÇÃO FORMAL DE UM PROTOCOLO DE REDE SEM FIO ATRAVÉS DE *MODEL CHECKING*

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2014

FABIANO ALMEIDA ROSAS

VERIFICAÇÃO FORMAL DE UM PROTOCOLO DE REDE SEM FIO ATRAVÉS DE *MODEL CHECKING*

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
Coorientador: Prof. MSc. Saulo Jorge Beltrão de Queiroz

PONTA GROSSA
2014

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa

Diretoria de Graduação e Educação Profissional

TERMO DE APROVAÇÃO

VERIFICAÇÃO FORMAL DE UM PROTOCOLO DE REDE SEM FIO ATRAVÉS DE MODEL CHECKING

por

FABIANO ALMEIDA ROSAS

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 11 de novembro de 2014 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
Prof. Orientador

_____
MSc. Saulo Jorge Beltrão de Queiroz
Membro titular

_____
Dra. Tânia Lúcia Monteiro
Membro titular

_____
Dr. Ionildo José Sanches
Responsável pelos Trabalhos
de Conclusão de Curso

_____
Dr. Gleifer Vaz Alves
Coordenador do Curso
UTFPR - Campus Ponta Grossa

A cópia do termo de aprovação assinada encontra-se disponível na coordenação do curso.

RESUMO

ROSAS, Fabiano Almeida. *Verificação formal de um protocolo de rede sem fio através de Model Checking*. 2014. 60 f. Trabalho de conclusão de curso (Bacharelado em Ciência da Computação) — Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2014.

O presente trabalho trata da aplicação da técnica de *model checking* na especificação formal, modelagem e verificação formal do método de acesso básico da função de coordenação distribuída (DCF) do protocolo de controle de acesso ao meio do padrão IEEE 802.11. A modelagem foi feita assumindo nós com tráfego saturado e com todas as estações iniciando o seu funcionamento ao mesmo tempo. O modelo proposto admite a configuração de parâmetros típicos de uma rede IEEE 802.11 a/b/g, tais como número de nós, largura de canal, taxa de transmissão e tamanho de pacote. Utilizou-se do *model checker* UPPAAL e verificou-se propriedades relacionadas ao tempo mínimo necessário para que todas as estações possam transmitir com sucesso. Este trabalho pretende preencher uma lacuna existente com relação à modelagem formal do IEEE DCF e busca servir de base para a verificação formal de novas variações do 802.11 DCF.

Palavras-chaves: verificação formal. model checking. UPPAAL. DCF. 802 11.

ABSTRACT

ROSAS, Fabiano Almeida. *Formal verification of a wireless network protocol through Model Checking*. 2014. 60 f. Trabalho de conclusão de curso (Bacharelado em Ciência da Computação) — Federal University of Technology - Paraná, Ponta Grossa, 2014.

This work consists of the application of the *model checking* technique to the formal specification, modeling and formal verification of the basic access method of the IEEE 802.11 standard medium access control protocol distributed coordination function (DCF). The modeling was made assuming nodes with saturated traffic and all stations starting at the same time. The purposed model admits typical configuration parameters of a IEEE 802.11 a/b/g network, such as number of nodes, channel width, data rate and packet size. Using the UPPAAL *model checker*, properties related to the minimum time needed for all the stations to transmit successfully were formally verified. This work intends to fill a gap that exists with relation to the formal modeling of the IEEE 802.11 DCF and serve as a basis for the formal verification of new 802.11 DCF variations.

Key-words: formal verification. model checking. UPPAAL. DCF. 802 11.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Conforme previsto por Yen e Chou (2001), a tecnologia *wireless* é hoje parte de nosso cotidiano. As redes sem fio, e mais especificamente, as redes locais sem fio (WLANs, do inglês *Wireless Local Area Network*) têm experimentado um aumento de popularidade com o passar dos anos devido à sua característica de permitir que a comunicação seja feita de maneira móvel (QADIR; AHMED; AHAD, 2013; KWIATKOWSKA; NORMAN; SPROSTON, 2002).

Com relação às WLANs, uma questão importante de seu funcionamento é que se houverem múltiplas transmissões ao mesmo tempo, poderá haver perda de dados, o que é conhecido como colisão. Para evitar colisões, é necessária a aplicação de estratégias de controle de acesso ao meio (*Medium Access Control* - MAC)(GUMMALLA; LIMB, 2000).

O padrão 802.11 da *Institute of Electrical and Electronics Engineers* (IEEE) define como estratégia MAC a função de coordenação distribuída (DCF) que é baseada no método *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA) (XU; SAADAWI, 2001).

Apesar da importância que protocolos como o IEEE 802.11 DCF apresentam para o pleno funcionamento de uma rede de computadores, há uma defasagem no que diz respeito ao rigor utilizado no projeto e implementação de tecnologias. Com a rápida evolução da *Internet*, as aplicações e protocolos de rede foram sendo desenvolvidas de uma maneira dinâmica levando a uma cultura onde, segundo Qadir e Hasan (2013), velocidade, heurísticas e código "rodando" são mais valorizadas do que conceitos sólidos de engenharia ou verificações formais.

Para a definição de sistemas, as especificações utilizadas geralmente são expressas através de linguagem natural, figuras e diagramas, pois precisam ser intuitivas e de fácil entendimento para que possam ser interpretadas e implementadas com facilidade. Isto faz com que as especificações acabem permitindo um certo grau de ambiguidade, porque figuras e diagramas podem ser interpretados de formas diferentes (MÜFFKE, 2004).

A utilização de métodos formais na verificação de protocolos de comunicação possui o benefício de adicionar rigor matemático às etapas do processo de construção e manutenção de um protocolo. As especificações formais, por exemplo, que são descrições de um sistema feitas através de uma linguagem lógica podem servir como interface de comunicação entre os diversos profissionais envolvidos no processo de desenvolvimento (BAIER; KATOEN, 2008).

Uma técnica que é utilizada na verificação de sistemas computacionais como protocolos de rede é o *model checking*, que alia a exploração exaustiva do espaço de estados (o que é realizado há algum tempo por técnicas de métodos formais, *e.g.* as baseadas em redes de petri), à definição de propriedades em lógica temporal, fazendo com que seja uma técnica conveniente para tratar de problemas como concorrência e eventualidade (CLARKE, 2008).

Inicialmente, o objetivo deste trabalho era utilizar-se da técnica de *Model checking* para fazer a verificação formal do protocolo PbP-DCF - que é uma generalização da estratégia de controle de acesso ao meio do padrão IEEE 802.11 (DCF) - que busca aumentar a vazão do

padrão através da utilização de múltiplas transmissões em canais secundários. Verificou-se, no entanto, a ausência de estudos formais sobre o funcionamento do próprio IEEE 802.11 DCF (conforme exposto na seção 1.3, a seguir), o que levou à readequação do objetivo deste trabalho, ficando a verificação formal do protocolo PbP-DCF como uma proposta para trabalhos futuros.

Neste contexto, o escopo deste trabalho consiste da aplicação de métodos formais, mais especificamente, da técnica de *model checking* visando a construção de um modelo formal para o IEEE 802.11 DCF que capture parâmetros relevantes de configuração de redes (*e.g.* largura de canal, taxa de transmissão) e que sirva de suporte à verificação formal de outros protocolos derivados do IEEE 802.11 DCF, como o PbP-DCF.

1.1 OBJETIVOS

Os objetivos deste trabalho são:

- Geral
  Efetuar, através de técnicas e ferramentas de *model checking*, a verificação formal do protocolo IEEE 802.11 DCF.
- Específicos
  - **i)** levantar o conjunto de propriedades que caracteriza o funcionamento adequado de um protocolo de acesso ao meio para redes locais sem fio;
  - **ii)** modelar, através de ferramentas de *model checking*, o funcionamento do protocolo;
  - **iii)** especificar formalmente, as propriedades de interesse utilizando-se de uma linguagem baseada na lógica temporal;
  - **iv)** verificar, através de técnicas de *model checking*, que o modelo atende à sua especificação.

1.2 JUSTIFICATIVA

As redes sem fio representam uma parte importante da infraestrutura de comunicação atual e o seu crescimento e popularização se deu de forma acelerada nos últimos anos (FRUTH, 2011). Apesar disto, os formalismos para modelagem e análise de redes sem fio não tiveram o mesmo desenvolvimento e, de acordo com Chalmers *et al.* (2006), "existe uma considerável falta de fundamentos formais" nesta área.

Ao se considerar uma tecnologia em particular, como protocolos de rede sem fio, percebe-se que a situação não é diferente e a modelagem é feita através de métodos que possuem um certo grau de ambiguidade. Quanto à verificação, são utilizadas técnicas baseadas em simulação que

possuem problemas conhecidos como, por exemplo, a variação de resultados entre simuladores diferentes (MÜFFKE, 2004).

Deve-se, ainda, levar em conta que protocolos são elementos de grande importância em redes e sistemas distribuídos e falhas ou ausência de comunicação podem, potencialmente, levar a situações de prejuízos financeiros, como no caso de bancos ou a perdas de vidas humanas, como em hospitais ou sistemas de transporte (SIMOVNÁK, 2012). Qadir e Hasan (2013) afirmam que, no mundo atual, é imperativo que os métodos de verificação manuais - sujeitos a erros - sejam evitados e que se automatize ao máximo as tarefas de verificação.

Tendo em vista estes fatos, ressalta-se ainda, que comparada com outras técnicas de verificação formal, como prova automática de teoremas ou checagem de provas, o *model checking* possui algumas vantagens como:

- Não há a necessidade de se construir provas de corretude;
- Do ponto de vista do especificador, *model checking* é mais rápido em comparação com as outras técnicas;
- Se uma especificação não for satisfeita, o *model checker* devolve uma execução de contra-exemplo. Clarke (2008) ressalta que algumas pessoas utilizam-se da técnica de *model checking* apenas para tirar proveito desta característica;
- Especificações podem ser parciais, ou seja, é possível utilizar o *model checking* durante o projeto de sistemas grandes sem a necessidade de esperar o término da fase de projeto;
- O uso de lógica temporal permite expressar com facilidade muitas das propriedades que são necessárias para o raciocínio sobre problemas concorrentes.

Além disso, o trabalho aqui proposto pretende aplicar a técnica de *model checking* em um protocolo que ainda não possui um estudo formal detalhado.

1.3 TRABALHOS RELACIONADOS

A referência mais antiga relacionada a este trabalho é o artigo de Bochmann (1978) em que é apresentada uma visão geral do uso de máquinas de estado finito para a verificação de programas. Neste artigo, também, Bochmann sugere o uso da "abstração de meio vazio"[1] para a redução da complexidade do sistema modelado. Ainda pelo mesmo autor, em 1980 foi apresentada uma visão geral do processo de especificação de um protocolo de rede com a primeira definição de alguns termos utilizados até os dias de hoje. Além disso, técnicas para lidar com a explosão do espaço de estados foram sugeridas.

[1] citada neste trabalho na Seção 2.3

Os dois artigos que iniciaram a área de *model checking* foram publicados independentemente, em 1982, por Edmund Clarke & Allen Emerson e Sifakis. Evoluções no estado da arte foram posteriomente feitas quando Vardi e Wolper (1986) sugeriram uma abordagem baseada em teoria de autômatos para a verificação de programas, partindo do princípio, demonstrado pelos autores, que uma fórmula de lógica temporal pode ser convertida em um autômato finito.

A aplicação de verificação formal aos protocolos de controle de acesso ao meio foi explorada através do uso da técnica de *model checking* probabilístico por Kwiatkowska, Norman e Sproston (2002) com a formalização do protocolo CSMA/CA. Os autores modelaram um sistema não-reativo (*i.e*, existe um estado terminal) e verificaram propriedades de cunho probabilístico relacionadas a colisões para uma rede com dois nós.

Em 2007, Clarke, Emerson e Sifakis foram prestigiados com o Turing Award pelo desenvolvimento do *model checking*. Em sua Turing Lecture, os autores apresentam uma visão geral do processo de *model checking* na época (CLARKE; EMERSON; SIFAKIS, 2009).

Mais recentemente, em 2011, Matthias Fruth aplicou a técnica de *model checking* probabilístico na verificação do processo controle de acesso ao meio do padrão 802.15.4 - usado para redes de sensores - de forma a analisar características de desempenho e consumo de energia. Para diminuir o tamanho do espaço de estados, Fruth modela dois nós e o envio de apenas uma mensagem além de utilizar-se de uma abstração da escala de tempo para reduzir o valor das constantes envolvidas. A propriedades verificadas foram:

- Probabilidade dos dois nós completarem sua transmissão;
- Máxima probabilidade de haver pelo menos k colisões;
- Máximo número esperado de colisões até que os dois nós tenham transmitido com sucesso;
- Tempo máximo esperado até que os dois nós tenham transmitido com sucesso;
- Energia máxima consumida até que os dois nós tenham transmitido com sucesso.

Além disso, foi feita uma meta-análise das decisões de modelagem aplicadas no estudo.

Ainda com relação ao padrão 802.15.4, Bulychev *et al.* (2013) realizaram a sua modelagem com suposições similares às de Fruth, porém sem a abstração da escala de tempo. Contudo, este modelo é referenciado no artigo para outros fins que não o *model checking*, logo não há mais informações sobre o mesmo.

Finalmente, Qadir e Hasan (2013) realizaram uma *survey* bastante detalhada da área de *model checking* aplicado à área de redes. Os autores sugerem a aplicação do *model checking* na área de redes definidas por software.

Como pode ser observado pelos trabalhos supra-citados existe uma lacuna com relação à aplicação da técnica de *model checking* à verificação formal de protocolos de controle de acesso ao meio, já que o padrão 802.15.4 possui certas particularidades, principalmente com relação ao controle do gasto de energia, inexistentes no 802.11, cuja verificação formal é aqui apresentada.

1.4 ESTRUTURA DO DOCUMENTO

O restante deste documento está estruturado da seguinte forma. No capítulo 2 apresenta-se conceitos de métodos formais, descrevendo-se as suas duas partes principais: especificação e verificação formais. O capítulo 3 trata da técnica de *model checking* e da ferramenta utilizada na execução do trabalho. No capítulo 4 descreve-se, de formas gerais, o protocolo CSMA/CA e no capítulo 5 o método empregado e o desenvolvimento do trabalho. Por fim, no capítulo 6 estão as considerações finais.

2 MÉTODOS FORMAIS

Métodos formais são técnicas baseadas na lógica e matemática e usadas na computação para descrever propriedades de sistemas. Diz-se que uma técnica é formal se ela possui uma base matemática sólida, tipicamente expressa na forma de uma linguagem de especificação que possui uma semântica formal que permite a definição de especificações sem ambiguidade (WOODCOCK *et al.*, 2009; WING, 1998; PLAT; KATWIJK; TOETENEL, 1992).

O rigor matemático presente nas técnicas de métodos formais permite revelar ambiguidades, incompletude e inconsistências em qualquer etapa do desenvolvimento de um *software* (WOODCOCK *et al.*, 2009):

Engenharia de requisitos: Métodos formais são utilizados para elicitar, articular e representar requisitos. Podem ser utilizados também para garantir completude e apoiar a evolução dos requisitos;

Especificação: Usados para expressar de maneira precisa o que o *software* deve fazer e quais são as restrições que devem ser obedecidas para tal;

Arquitetura: No caso de sistemas complexos, é necessária uma organização cuidadosa da estrutura dos componentes. Modelos formais podem prover uma abstração conveniente, escondendo detalhes da implementação para que os arquitetos possam focar em características mais relevantes;

Projeto: No projeto de *software* a aplicação de métodos formais pode se dar na forma da construção de máquinas de estados, funções abstratas e simulações;

Implementação: Neste nível, métodos formais são utilizados na verificação de código. Um trecho de código pode ser verificado de acordo com uma especificação para garantir corretude e conformidade;

Teste e Manutenção: Nestas duas fases de um projeto, métodos formais têm um papel inerente principalmente no uso de asserções. Woodcock *et al.* (2009) citam o uso de asserções na verificação de *software* legado na *Microsoft*, reportado por Hoare em 2002 .

Métodos formais servem, então, para provar que uma especificação é possível de ser implementada, provar que uma implementação está correta ou provar propriedades de um sistema sem que haja a necessidade de executá-lo (WING, 1998).

O entendimento dos métodos formais se faz através de dois conceitos principais: especificação formal e verificação formal.

2.1 ESPECIFICAÇÃO FORMAL

A especificação é o processo de descrição de um sistema e das propriedades que se espera que ele tenha. A especificação **formal** é uma especificação feita utilizando-se de uma linguagem formal que provê a base matemática do método formal (CLARKE; WING, 1996).

As linguagens de especificação formal provêm uma notação (domínio sintático), um universo (domínio semântico) e uma regra definindo quais objetos do universo satisfazem a especificação. Os métodos formais diferem devido ao fato de suas linguagens de especificação formal possuírem domínios sintáticos e semânticos diversos (WING, 1998).

2.1.1 Domínio Sintático

O domínio sintático de uma linguagem é definido por um conjunto de símbolos como constantes, variáveis, operadores lógicos e um conjunto de regras gramaticais para formar sentenças com estes símbolos. Um exemplo seriam os símbolos $\forall \Rightarrow P\ Q$ que podem ser combinados para formar a expressão da lógica de predicados: $\forall$x.P(x) $\Rightarrow$ Q(x).

2.1.2 Domínio Semântico

No domínio semântico, os símbolos e expressões do domínio sintático são usados para dar sentido à especificação. Alguns exemplos são as linguagens de especificação de:

Tipos abstratos de dados - utilizadas para especificar álgebras, teorias e programas;

Sistemas distribuídos e concorrentes - especificam sequências de estados, sequências de eventos e máquinas de estados;

Programas lineares - especificam funções que levam uma entrada à uma saída, computações e instruções de máquina.

A utilização de uma linguagem de especificação com este nível de formalismo leva a uma descrição do sistema de forma precisa e sem ambiguidades (WING, 1998). O maior benefício deste processo é não apenas a obtenção de um maior entendimento do sistema, levando ao descobrimento de falhas, inconsistências e incompletudes, mas também a geração de um produto (especificação) que pode ser utilizado como meio de comunicação entre as partes envolvidas no projeto. Além disso, a especificação formal, como um artefato, pode ser verificada formalmente de maneira mecânica através de técnicas e ferramentas de verificação formal (CLARKE; WING, 1996).

2.2 VERIFICAÇÃO FORMAL

A verificação formal é utilizada para averiguar se o produto ou projeto em questão possui certas propriedades. A especificação formal é a base deste processo e serve para prescrever o que o produto deve e não deve fazer. Se um sistema estiver em conformidade com todas as propriedades especificadas, diz-se que este sistema é correto. Desta forma, corretude é uma característica relativa a uma especificação (BAIER; KATOEN, 2008). A Figura 1 mostra o processo de verificação formal a partir da especificação do sistema. Após a fase de projeto, um produto ou protótipo será criado e servirá, juntamente com as propriedades que foram levantadas a partir da especificação, como entrada para o processo de verificação formal. Desta forma, verifica-se a conformidade do produto com as propriedades desejadas.

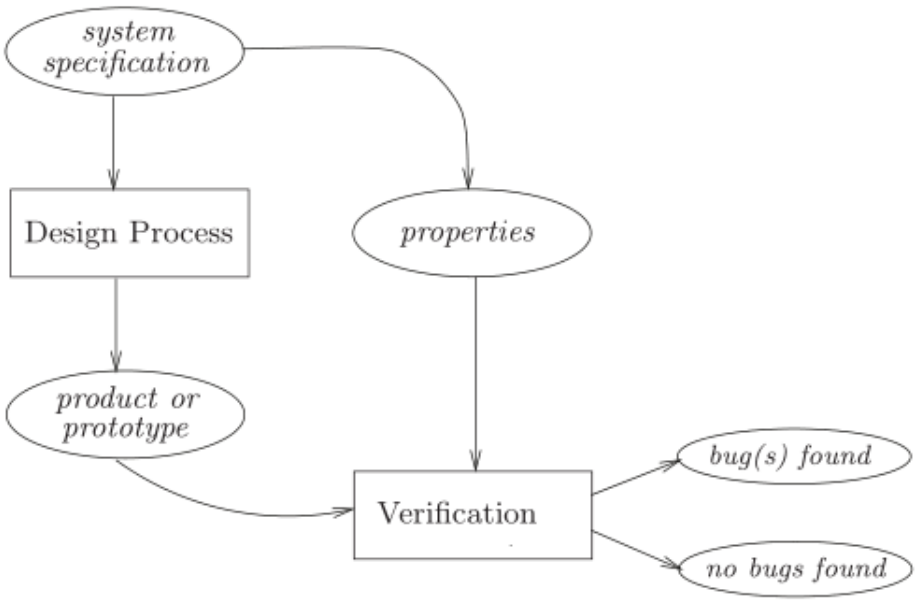


Figura 1 – O processo de verificação formal. Um *produto* é verificado com respeito a um conjunto de *propriedades* para checar sua corretude

Fonte: Baier e Katoen (2008, pg. 3)

As duas formas principais de verificação formal são a *prova de teoremas* e o *model checking*. A *prova de teoremas* é uma técnica na qual tanto o sistema quanto suas propriedades são expressos em uma linguagem lógica matemática. Como a linguagem lógica faz parte de um sistema formal composto de regras de inferência e axiomas, pode-se verificar através de provas lógicas se a especificação corresponde às propriedades. O *model checking* é uma técnica que se baseia na construção de um modelo e a verificação é feita através de uma análise exaustiva no espaço de estados deste modelo (CLARKE; WING, 1996).

2.3 O PROBLEMA DA EXPLOSÃO DE ESTADOS

O problema da explosão de estados foi primeiro definido em Bochmann e Sunshine (1980) e diz respeito à impossibilidade de se verificar um modelo devido ao tamanho do espaço global de estados que, durante a verificação de um sistema distribuído como um protocolo de rede, pode crescer exponencialmente, devido ao grande número de entidades (processos ou módulos) que interagem entre si durante a execução do sistema. As técnicas sugeridas para lidar com este problema são:

Verificação parcial: dependendo do método de verificação utilizado, apenas alguns aspectos do sistema podem ser modelados, deixando alguns detalhes (que não são considerados fundamentais) de lado.

Uso de unidades de abstração grandes: o problema da explosão de estados se dá devido à intercalação de ações de entidades diferentes. Ao se escolher unidades de abstração grandes, diminui-se o número de ações que podem ser intercaladas, diminuindo também o espaço de estados. Um exemplo deste processo é quando se utiliza de uma "abstração de meio vazio", ou seja, considera-se que nunca há dados em trânsito em um meio de transmissão. No caso de troca de mensagens, isso implica em modelar o envio juntamente com o recebimento de uma mensagem, o que garante que não haverá ações sendo intercaladas entre o momento em que o remetente envia e o receptor recebe a mensagem.

Decomposição em subfases: ao se decompor um sistema em subfases, ocorre uma simplificação do processo e cada uma das fases pode ser verificada separadamente, não adicionando ao número total de estados.

Classificação de estados: a classificação de estados através de asserções permite que vários estados sejam tratados em um conjunto. Por exemplo, quando uma entidade recebe *frames* de informação numerada, basta apenas que se classifique os estados em maior que, igual a e menor que o número de *frames* que se espera receber. Todos os outros estados (1 unidade maior, 3 unidades menor, etc...) podem ser ignorados.

Busca direcionada: ao invés de se gerar todos os estados do sistema, faz-se uma análise prévia e determina-se quais são os estados de interesse (que dão início a um *deadlock*, por exemplo) e modela-se apenas estes estados.

Automação: muitas etapas da verificação podem ser delegadas a programas que executem algumas etapas da análise. O *model checking* que será tratado no capítulo 3 é um exemplo de forma de automatizar o processo de verificação.

2.4 A NECESSIDADE DE MÉTODOS FORMAIS

Sistemas complexos de *hardware* e *software* são atualmente utilizados em aplicações onde erros não são toleráveis. Estes sistemas são altamente acoplados ao nosso cotidiano como, por exemplo, sistemas bancários e telefones celulares. Outros são utilizados em situações de alto risco, como em ambientes hospitalares ou em torres de controle de tráfego aéreo (CLARKE JR.; GRüMBERG; PELED, 1999).

A literatura cita alguns exemplos clássicos onde a falta de formalização em um sistema computacional causou grandes prejuízos a companhias e indivíduos. Clarke Jr., Grümberg e Peled (1999) descrevem o incidente com foguete Ariane 5, em 1996, que explodiu menos de 40 segundos após o lançamento devido a uma *exceção* ocorrida na conversão de um número em ponto flutuante de 64 bits para um inteiro de 16 bits. Baier e Katoen (2008) relatam o conhecido caso do *bug* presente na unidade de divisão de números em ponto flutuante do Pentium II da Intel que no início dos anos 90 causou à empresa perdas de 475 milhões de dólares. Outro fato, relatado no mesmo livro, foi o atraso de 9 meses na inauguração do aeroporto de Denver devido a falhas no *software* de movimentação de bagagens que trouxe perdas de 1,1 milhão de dólares por dia.

Além dos exemplos dados, convém citar aqui a importância dos sistemas distribuídos que têm sofrido um aumento em pervasividade e complexidade devido ao desenvolvimento dos sistemas em nuvem. Garantir a qualidade destes sistemas é uma tarefa não-trivial já que dependem de fatores imprevisíveis, *e.g* velocidade de processamento dos componentes e taxas de transmissão da rede (DIN *et al.*, 2012; DIN; OWE; BUBEL, 2014). Estes fatores, quando somados à crescente pressão para que se reduza o tempo de desenvolvimento de um sistema devido a fatores do mercado faz com que a construção de sistemas de informação e comunicação sem defeitos seja algo desafiador (BAIER; KATOEN, 2008).

O uso de métodos formais, embora possa não fornecer uma garantia completa de corretude, pode auxiliar no entendimento detalhado de um sistema, revelando inconsistências que passariam despercebidas (CLARKE; WING, 1996).

2.5 MÉTODOS FORMAIS NA VERIFICAÇÃO DE PROTOCOLOS DE REDE

Durante o seu desenvolvimento, protocolos precisam ser descritos em várias situações. O projeto inicial precisa ser descrito para facilitar a cooperação entre projetistas, bem como checado para garantir corretude. Além disso o protocolo precisa ser implementado e, caso várias implementações existam, elas precisam ser checadas para garantir que estejam em conformidade com o projeto inicial (BOCHMANN; SUNSHINE, 1980). Cada uma destas etapas gera riscos para a introdução de erros e inconsistências. O processo tradicional de especificação de

um protocolo adotado pela *Internet Engineering Task Force* (IETF) é baseado em especificações escritas utilizando linguagem natural e baseando-se em implementações como um auxílio informal (QADIR; HASAN, 2013). Isto faz com que a distinção entre decisões de projeto e decisões de implementação em um sistema específico se torne pouco clara, o que agrava ainda mais a situação. Diante destes fatos, a aplicação de métodos formais no desenvolvimento de protocolos pode ser um fator benéfico.

3 MODEL CHECKING

O *model checking* é uma técnica de verificação formal que consiste em construir um modelo de um sistema e verificar se uma certa propriedade é válida neste modelo (CLARKE; WING, 1996). O problema do *model checking* é um caso particular do problema da verificação:

> *Dado um programa* M *e uma especificação* h*, determinar se o comportamento de* M *satisfaz* h*.*

No caso do *model checking*, o *model checker* (*software*) examina todos os estados relevantes de um sistema de estados finito e verifica se eles satisfazem uma determinada propriedade. Caso a busca chegue em um ponto onde a propriedade em questão foi violada, a ferramenta de *model checking* apresenta um contra-exemplo descrevendo qual o caminho percorrido até chegar àquele estado (CLARKE; WING, 1996; BAIER; KATOEN, 2008; EMERSON, 2008). Segundo Ernest Emerson, um dos criadores do *model checking*, "já que a maioria dos programas contêm erros, um importante ponto forte dos *model checkers* é que eles provêm um contra-exemplo para a maioria dos erros" (EMERSON, 2008).

O *model checking* surgiu como solução ao problema da verificação de programas concorrentes. Erros em programas concorrentes são difíceis de serem encontrados utilizando-se testes padrão, já que são difíceis de serem reproduzidos. Antes da criação do *model checking*, a verificação de programas concorrentes era feita através da construção manual de provas (CLARKE, 2008). A maioria dos problemas encontrados pelos *model checkers* são problemas clássicos de concorrência, como processos executando de maneira alternada e acessando variáveis compartilhadas (BAIER; KATOEN, 2008). Um exemplo básico desta situação é o exposto na Figura 2. Considere um sistema com os três processos descritos[1] na figura rodando em paralelo. O objetivo do sistema, conforme descrito informalmente, seria manter o valor da variável x no intervalo $[0, 200]$.

$$
\begin{aligned}
\textbf{proc}\ \mathsf{Inc} &= \textbf{while}\ \mathrm{true}\ \textbf{do if}\ x < 200\ \textbf{then}\ x := x + 1\ \textbf{fi od} \\
\textbf{proc}\ \mathsf{Dec} &= \textbf{while}\ \mathrm{true}\ \textbf{do if}\ x > 0\ \textbf{then}\ x := x - 1\ \textbf{fi od} \\
\textbf{proc}\ \mathsf{Reset} &= \textbf{while}\ \mathrm{true}\ \textbf{do if}\ x = 200\ \textbf{then}\ x := 0\ \textbf{fi od}
\end{aligned}
$$

Figura 2 – Três processos rodando de maneira concorrente

Fonte: Baier e Katoen (2008, pg. 10)

A princípio, este programa faz o que foi especificado, o processo Inc mantém a variável aumentando em direção a 200; Dec a mantém diminuindo em direção a 0; e Reset retorna x a 0 quando o valor 200 for atingido. O problema com o código acima, que provavelmente passou

[1] O código está escrito na linguagem Algol-68.

despercebido ao leitor, é o fato de que como os processos rodam em paralelo, de maneira concorrente com relação à variável x, pode ocorrer a seguinte situação: suponha que o valor de x em um dado momento é 200. O processo Dec efetua a comparação ($x > 0$) e o teste é avaliado para verdadeiro. Neste momento, o processo Reset também faz a sua comparação ($x = 200$) e como o teste também avalia para verdadeiro, prossegue com a execução do bloco e reinicia a variável x para 0. Dec, então, continua sua execução, e como o seu teste já havia passado, executa o seu bloco de código e decrementa x, colocando o sistema em um estado inesperado ($x = -1$) e potencialmente inconsistente e incorreto.

O *model checking* foi criado para ajudar a resolver problemas desta natureza e permitir um melhor entendimento dos programas concorrentes.

As principais formas de *model checking* são *model checking* temporal, no qual as especificações são feitas em lógica temporal e os sistemas são modelados como sistemas de transição de estados finitos; *model checking* baseado em autômatos, no qual as especificações são dadas como autômatos e os sistemas são modelados também como autômatos (CLARKE; WING, 1996); e uma abordagem híbrida de autômatos temporais que parte do princípio de que fórmulas da lógica temporal podem ser reescritas como autômatos finitos (VARDI; WOLPER, 1986).

Nas seções seguintes, apresenta-se uma descrição da lógica temporal e suas subdivisões (Seção 3.1); autômatos e autômatos temporais (Seções 3.2, 3.2.1) e as propriedades que são provadas através do *model checking* (Seção 3.3).

3.1 LÓGICA TEMPORAL

A corretude de sistemas reativos depende das possíveis execuções do sistema e não somente de suas entradas e saídas. A lógica temporal é um formalismo adequado para tratar estes aspectos, estendendo a lógica proposicional com modalidades que permitem raciocinar sobre o comportamento contínuo dos sistemas reativos (BAIER; KATOEN, 2008). O uso da lógica temporal para a verificação desse tipo de sistema foi primeiro proposto por Amir Pnueli em 1977 (PNUELI, 1977). Deste ponto em diante, várias extensões e modificações da lógica temporal têm sido criadas.

3.1.1 Lógica Temporal Linear

A lógica temporal linear (LTL) é uma extensão da lógica proposicional clássica com a adição de modalidades temporais básicas. Neste tipo de lógica, o tempo possui uma natureza linear, ou seja, cada momento é sucedido por apenas um único momento sucessor, enquanto que em lógicas ramificantes, cada momento pode ser sucedido por uma árvore de momentos sucessores (BAIER; KATOEN, 2008).

Os operadores da lógica temporal linear são, além dos conectivos booleanos de conjunção $\wedge$ e negação $\neg$, os operadores temporais seguinte $\bigcirc$ e até que $\mathcal{U}$. O operador $\bigcirc$ é unário e seu significado é *a fórmula $\bigcirc\varphi$ é válida neste momento se no momento seguinte φ for verdade*. O operador $\mathcal{U}$ é binário e seu significado é *a fórmula $\varphi_1 \mathcal{U} \varphi_2$ é válida se existe um momento futuro em que φ_2 é verdade e φ_1 é verdade em todos os momentos até aquele momento* (BAIER; KATOEN, 2008).

O operador $\mathcal{U}$ pode ser utilizado para obter os operadores eventualmente $\Diamond$ e sempre $\Box$ da seguinte forma:

$$\Diamond\varphi = true\,\mathcal{U}\,\varphi$$
$$\Box\varphi = \neg\Diamond\neg\varphi$$

A combinação destes dois operadores resulta nas construções $\Box\Diamond\varphi$ (infinitas vezes φ) e $\Diamond\Box\varphi$ (eventualmente para sempre φ).

Para exemplificar a utilização da lógica temporal linear na modelagem de sistemas reativos, pensemos em um sistema que controla um semáforo, com os estados "verde", "amarelo" e "vermelho". Algumas fórmulas que poderiam ser elaboradas para garantir a corretude deste sistema são:

$\Box\Diamond verde$ o semáforo se torna verde infinitas vezes;

$\Box(vermelho \rightarrow \neg\bigcirc verde)$ uma vez vermelho, o semáforo não pode se tornar verde logo a seguir;

$\Box(vermelho \rightarrow \bigcirc(vermelho\mathcal{U}(amarelo \wedge \bigcirc(amarelo\mathcal{U}\,verde))))$ uma vez vermelho, o semáforo sempre se torna verde eventualmente após estar amarelo por algum tempo.

3.1.2 Lógica de Árvore de Computação

Enquanto a LTL é expressa em termos de caminhos, *i.e.* um único futuro possível a partir de determinado ponto, a lógica de árvore de computação (*Computation Tree Logic* - CTL) é expressa em termos de uma árvore infinita de estados, o que significa que a qualquer momento, podem existir vários futuros possíveis (BAIER; KATOEN, 2008).

Os operadores temporais da CTL permitem expressar propriedades de *alguma* ou *todas* as computações que se iniciam em um estado através dos quantificadores $\exists$ e $\forall$, respectivamente. Note que $\exists\Diamond\varphi$ significa que *existe um caminho no qual φ eventualmente é verdade*, porém isto não significa que não possa haver outros caminhos em que φ é falso (BAIER; KATOEN, 2008).

A sintaxe da CTL é de certa forma mais simples do que a LTL, não permitindo a combinação de operadores temporais com conectivos booleanos e nem o encadeamento de operadores temporais. Uma fórmula CTL é composta de duas partes: uma que se refere aos estados ($\exists, \forall$) e

outra que se refere aos caminhos ($\bigcirc, \mathcal{U}$). Operadores de caminhos sempre devem ser precedidos de operadores de estados. Como na LTL, os operadores $\Diamond$ e $\Box$ podem ser derivados do operador $\mathcal{U}$ (BAIER; KATOEN, 2008).

Seguindo o mesmo exemplo do semáforo citado anteriormente, podemos expressar as seguintes propriedades:

$\forall\Box(amarelo \vee \forall\bigcirc\neg vermelho)$ todo estado vermelho é sempre precedido por um estado amarelo;

$\forall\Box\forall\Diamond verde$ o semáforo se torna verde infinitas vezes.

3.1.3 Lógica de Árvore de Computação com Tempo

Para melhor expressar situações envolvendo tempo-real, temos uma variação da CTL, a lógica de árvore de computação com tempo (*Timed Computation Tree Logic* - TCTL), que possui uma noção contínua de tempo, ao contrário da LTL e CTL que possuem apenas uma visão relativa. Na TCTL é possível expressar, por exemplo, que um estado é atingível após t unidades de tempo (BAIER; KATOEN, 2008). As fórmulas de TCTL são geralmente avaliadas levando em consideração um autômato temporal, que será visto em detalhes na seção 3.2.1.

Como na CTL, existem dois tipos de fórmulas, as de estado e as de caminho, que são combinadas para formar uma fórmula válida. A diferença é que na TCTL, além das proposições atômicas, um conjunto de relógios pode ser utilizado, onde um relógio é uma estrutura que conta o tempo (geralmente relacionado ao relógio de um autômato temporal), sendo representada por um número $j \subseteq \mathbb{R}_{\geq 0}$ com limites $\in \mathbb{N}$ (BAIER; KATOEN, 2008).

Os operadores da TCTL são os mesmos da CTL, com exceção do operador $\bigcirc$, que se torna impossível de definir, devido ao fato do tempo ser contínuo. Para exemplificar o uso dos operadores, considere o exemplo de uma lâmpada:

$\forall\Box(ligada \rightarrow \forall\Diamond^{>2}\neg ligada)$ a lâmpada não pode estar ligada por mais de 2 minutos;

$\forall\Box((ligada \wedge (x = 0)) \rightarrow (\forall\Box^{\leq 1} ligada \wedge \forall\Diamond^{>1} desligada))$ a lâmpada ficará ligada por pelo menos 1 minuto e então desligará.

Na fórmula acima, x é um relógio e representa o instante em que a lâmpada será ligada.

3.2 AUTÔMATO FINITO

Um grafo de transição de estados finito rotulado[2] pode ser tido como um autômato finito não-determinístico (*Non-deterministic finite automata - NFA*).

Definição 1 – Autômato finito não-determinístico

Um autômato finito não determinístico A é uma 5-tupla A = (Q, Σ, δ, q_0, F), onde:

- Q é um conjunto finito de estados,
- Σ é o alfabeto do autômato,
- $\delta : Q \times \Sigma \rightarrow 2^Q$ é a função de transição,
- $q_0 \in$ Q é o estado inicial e
- F $\subseteq$ Q é o conjunto de estados finais.

A Figura 3 mostra a representação gráfica de um NFA.

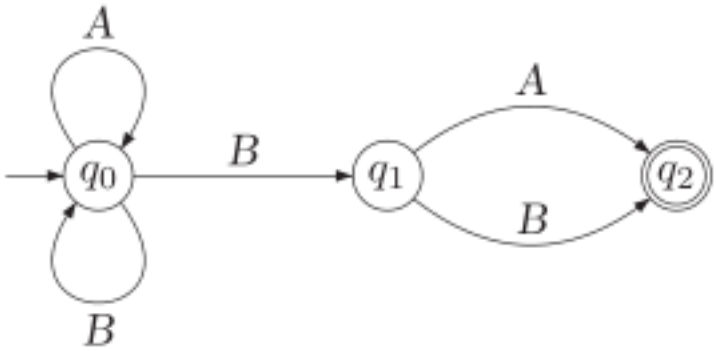


Figura 3 – Autômato finito não-determinístico

Fonte: Baier e Katoen (2008, pg. 14)

A relação entre um autômato finito e a lógica temporal foi estabelecida por Vardi e Wolper (1986), ao provarem que para cada fórmula da lógica temporal, é possível construir um autômato que aceite precisamente a mesma computação que satisfaz a fórmula. Neste momento também, foi traçada a ligação entre a verificação de programas concorrentes com a lógica temporal ao se propor que um programa fosse modelado de forma que cada um dos seus estados representasse um conjunto de proposições atômicas. Desta forma, uma computação passou a ser interpretada como uma palavra infinita sobre o alfabeto dos valores verdade das proposições atômicas a ser aceita ou rejeitada por um autômato finito (VARDI; WOLPER, 1986).

Embora um autômato finito possa ser utilizado para representar um sistema reativo com cada estado representando um estado do sistema e cada transição representando condições que

[2] Na área de *model checking*, um "grafo de transição de estados finito rotulado" é comumente chamado de "estrutura de Kripke". Segundo Browne, Clarke e Grümberg (1988), isto se dá por razões históricas.

levam de um estado a outro, esta representação apenas serve para mostrar como um sistema evolui de um estado a outro, deixando aspectos temporais de lado, ou seja, não há formas de indicar quanto tempo o sistema permanece em um estado e nem quanto tempo levará até que determinada transição seja feita (BAIER; KATOEN, 2008). Para tais afirmações, precisamos recorrer à noção de autômato temporal.

3.2.1 Autômato Temporal

O autômato temporal é uma representação que serve para modelar o comportamento de sistemas críticos em relação ao tempo e consiste de um autômato com um conjunto especial de variáveis chamadas relógios. Estes relógios possuem valores reais e marcam a passagem do tempo, sendo incrementados de maneira síncrona conforme o tempo avança. Relógios podem apenas ser zerados ou verificados (BAIER; KATOEN, 2008).

As transições de um autômato temporal, chamadas ações podem se encontrar ativas ou inativas, dependendo de determinadas condições previamente estabelecidas. Estas condições são chamadas de *guards* e, mais especificamente quando levam em consideração os relógios, de restrições de relógio (*Clock constraints*). As restrições de relógio também servem para determinar quanto tempo o sistema pode permanecer em um estado (BAIER; KATOEN, 2008).

A definição formal de um autômato temporal é apresentada na definição 2 (BEHRMANN *et al.*, 2002).

Definição 2 – Autômato temporal

Seja C um conjunto de relógios e $x, y \in C$; $c \in \mathbb{N}$; $op\_rel \in \{\leq, \geq, =, <, >\}$; define-se $B(C)$ como um conjunto de *guards*, onde *guard* é uma conjunção da forma $x\ op\_rel\ c$ ou $x - y\ op\_rel\ c$.

Um autômato temporal é uma 6-tupla (L, l_0, C, A, I, E), onde:

- L é um conjunto de *locations*, ou estados.
- $l_0 \in L$ é a *location* inicial.
- C é o conjunto de relógios.
- A é um conjunto de ações.
- $I : L \rightarrow B(C)$ uma função que atribui invariantes a *locations*.
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ é um conjunto de arestas entre duas *locations*, com uma ação, um *guard*, e um conjunto de relógios.

3.3 PROPRIEDADES

Quando interpretamos uma lógica temporal com base em um autômato temporal, certas propriedades podem ser extraídas (CLARKE, 2008). Propriedades, neste contexto, são asserções simples que dizem que um predicado das variáveis do programa é verdade quando a computação atinge um determinado ponto de controle (JHALA; MAJUMDAR, 2009). Alguns tipos de propriedades básicas são enumeradas nas seções seguintes.

3.3.1 Segurança

Estas são as propriedades da forma "algo ruim não irá acontecer". Utilizando a notação da lógica temporal, podemos dizer que este tipo de propriedade é expressa com as fórmulas lógicas $\forall\Box\varphi$ ou $\exists\Box\varphi$ (BAIER; KATOEN, 2008; BEHRMANN; DAVID; LARSEN, 2004). Exemplos desta classe são $\forall\Box\neg deadlock$ que significa que para todos os estados nunca haverá um *deadlock*; e $\forall\Box(\neg\text{seção-crítica}_1 \vee \neg\text{seção-crítica}_2)$ que expressa exclusão mútua.

3.3.2 Progresso

As propriedades de progresso, ou não-terminação significam que "algo irá acontecer". Este tipo de propriedade serve para complementar as propriedades de segurança, evitando assim que um sistema que não execute, ou que nunca chegue a certo ponto, seja considerado correto, já que estando estagnado, satisfaz as propriedades de segurança (BAIER; KATOEN, 2008). Para representar progresso, podemos utilizar a forma $\forall\Diamond\phi$, por exemplo $enviarMsg \Rightarrow \forall\Diamond receberMsg$ significa que todo envio de mensagem implica no recebimento de uma mensagem. Duas propriedades de progresso muito utilizadas são:

- Cada processo eventualmente executará uma ação;
- Cada processo irá executar a sua ação infinitas vezes.

A combinação destas duas propriedades garante a ausência de *starvation*.

3.3.3 Alcance

Estas são as mais simples das propriedades, que definem somente se algo *pode* acontecer. São geralmente utilizadas para conferir se o modelo está funcionando corretamente. São

do tipo $\exists \Diamond \phi$, como $\exists \Diamond \mathrm{enviarMsg}$ que expressa a possibilidade de enviar uma mensagem (BEHRMANN; DAVID; LARSEN, 2004).

A Figura 4 dá uma representação gráfica das propriedades descritas acima.

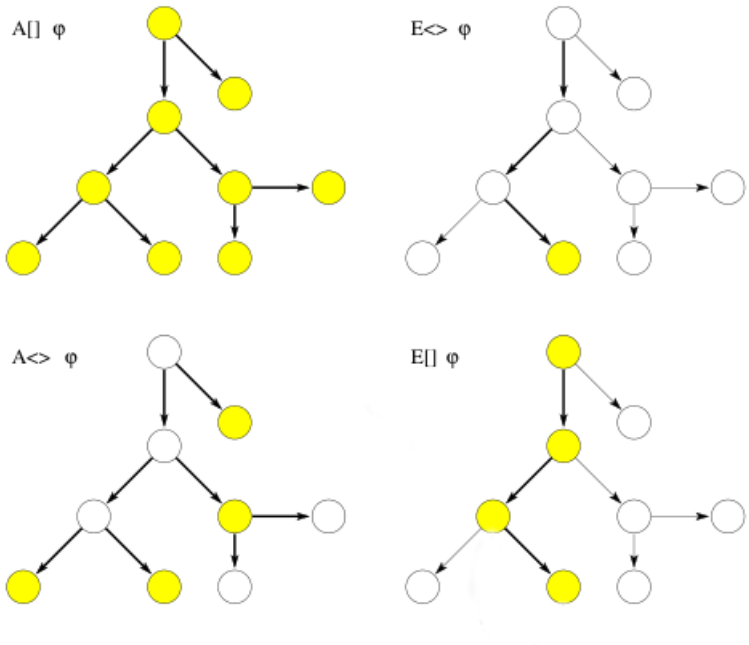


Figura 4 – Propriedades da lógica temporal expressas em autômato finito não-determinístico. À esquerda e acima e à direita e abaixo, segurança; à direita e acima, alcance; à esquerda e abaixo, progresso.

Fonte: Behrmann, David e Larsen (2004, pg. 8)

3.4 O PROCESSO DE *MODEL CHECKING*

O processo que se utiliza para verificar programas através de *model checking* pode ser dividido em três fases principais: **modelagem**, **execução** e **análise** (BAIER; KATOEN, 2008).

3.4.1 Modelagem

Na etapa de modelagem é necessário criar o modelo do sistema através da linguagem de modelagem da ferramenta que se pretende utilizar para fazer a verificação. Em alguns casos, isto é apenas uma tarefa de transcrição, enquanto que em outros, técnicas de abstração devem ser utilizadas para eliminar detalhes irrelevantes (CLARKE JR.; GRüMBERG; PELED, 1999).

Na construção de um modelo (na maioria dos casos, um autômato finito), os estados representam variáveis do sistema e as transições a forma como o sistema passa de um estado para outro (BAIER; KATOEN, 2008).

Para garantir que o modelo representa o sistema modelado adequadamente, uma etapa intermediária de simulação é realizada para eliminar os erros mais simples que podem ocorrer. Baier e Katoen (2008) ressaltam a importância desta etapa de simulação: “Qualquer verificação que se utilize de técnicas baseadas em modelos é apenas tão boa quanto o modelo criado.”

Ainda durante a fase de modelagem, é realizada a especificação das propriedades através de alguma linguagem lógica (comumente, lógica temporal), sendo que nesta etapa, é importante estar atento à garantia de completude da especificação, já que o *model checking* consegue verificar se um sistema atende à especificação, porém determinar se a especificação cobre todas as propriedades que se deseja que o sistema tenha está fora do escopo da técnica (CLARKE JR.; GRüMBERG; PELED, 1999).

3.4.2 Execução

A execução do *model checker*, que é a verificação, propriamente dita, ocorre idealmente de maneira automática, porém, na prática verifica-se que são necessários alguns ajustes e configurações do *model checker* para a obtenção de melhores resultados (CLARKE JR.; GRüMBERG; PELED, 1999; BAIER; KATOEN, 2008). Dependendo da ferramenta utilizada, algumas opções estão presentes:

Redução de ordem parcial - técnica que explora o fato de que em um sistema concorrente, alguns eventos podem acontecer de maneira independente uns dos outros, levando à possibilidade da redução do espaço de estados a ser verificado.

Redução de simetria - baseia-se no fato de que sistemas concorrentes geralmente possuem componentes repetidos, como processos idênticos que se comunicam em uma rede. Esta técnica serve para eliminar estes componentes repetidos de forma a minimizar o grafo de estados.

3.4.3 Análise

Após efetuada a execução do *model checker*, segue-se, naturalmente, para a análise do resultado obtido. Existem três resultados que podem ser obtidos. Caso a propriedade especificada seja dada como válida, pode-se seguir com a verificação das propriedades subsequentes. Caso a propriedade seja inválida, o *model checker* irá prover um contra-exemplo diagnóstico, que deverá então ser analisado através de simulação para que se encontre o ponto de falha do sistema. Além destes dois casos, um terceiro ainda é possível (mais frequente em sistemas maiores), que é quando o modelo não pode ser verificado devido a explosão do espaço de estados (BAIER; KATOEN, 2008). Neste último caso, o *model checker* (pessoa) deverá se utilizar das opções de

verificação citadas na fase anterior ou ainda alterar o modelo fazendo uso de algumas heurísticas:

Raciocínio composicional - como protocolos e programas complexos geralmente possuem uma estrutura modular, é possível dividir o sistema em partes menores que são verificadas individualmente. Em casos em que há dependências entre as partes, é possível, ainda, utilizar de uma estratégia chamada *assume-guarantee reasoning* que significa fazer a modelagem de uma parte presumindo que as outras partes das quais ela depende estão corretas (CLARKE JR.; GRüMBERG; PELED, 1999).

Abstração - a abstração pode ser empregada quando se observa que a especificação do sistema inclui relações simples entre dados, como por exemplo, o já citado (Seção 2.3) exemplo de uma variável que não precisa necessariamente ter valores reais, apenas valores relativos a outra variável (maior que, menor que, etc...). Utilizando a abstração desta forma, é possível produzir sistemas abstratos muito menores do que suas contrapartes (CLARKE JR.; GRüMBERG; PELED, 1999).

Sistemas parametrizados - protocolos de rede são em geral sistemas parametrizados, ou seja, fazem parte de uma família maior de processos. Nestes casos, uma técnica a ser empregada é a construção de um sistema invariante, de tamanho menor, que represente todos os membros de uma família (CLARKE JR.; GRüMBERG; PELED, 1999).

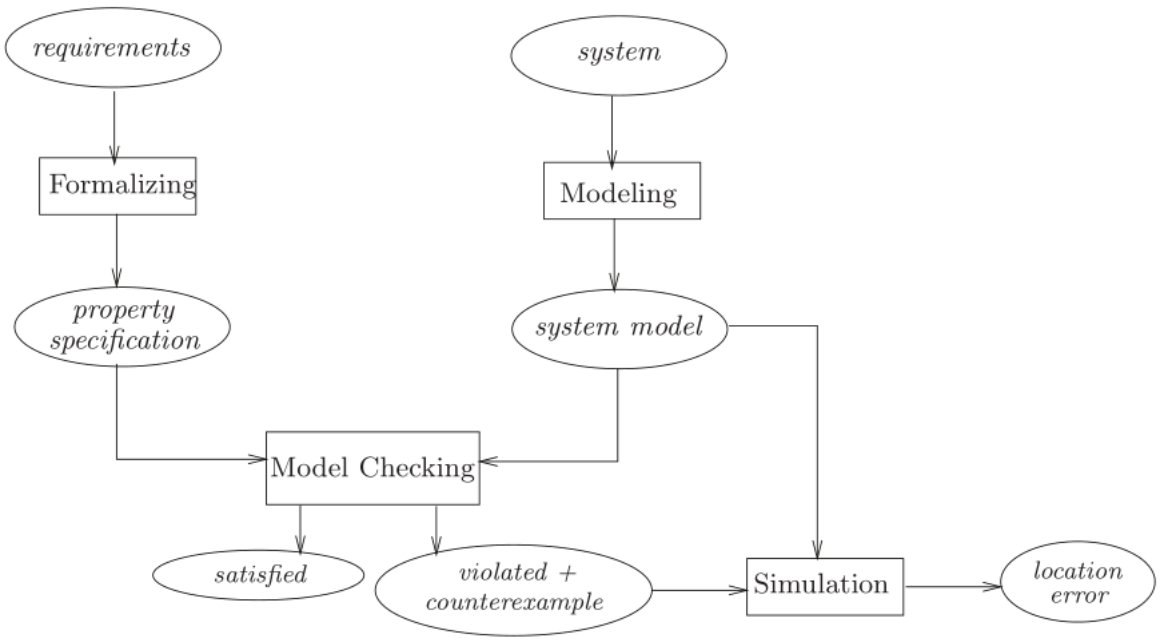


Figura 5 – Visão geral do processo de *model checking*

Fonte: Baier e Katoen (2008, pg. 8)

A Figura 5 ilustra o processo de *model checking* em sua totalidade: requerimentos são formalizados e um sistema é modelado, servindo tanto a especificação de propriedades quanto o modelo do sistema como entrada para o *model checker*. Ao final do processo, o *model checker*

responderá se as propriedades foram satisfeitas ou violadas, caso no qual um contra-exemplo será dado para análise, geralmente através de um simulador que servirá para encontrar em que ponto da execução ocorreu o problema.

3.5 VANTAGENS E DESVANTAGENS

A seguir, estão listadas algumas das vantagens e desvantagens do *model checking* com relação às outras técnicas de verificação como a prova de teoremas, simulação e teste.

Segundo Clarke (2008) e Baier e Katoen (2008):

Vantagens:

- Não é necessário fazer provas.
- É uma técnica rápida.
- Produz contra-exemplos.
- Permite especificações parciais.
- A lógica temporal tem boa expressividade dos conceitos de programas concorrentes.
- Processo generalista com uma variedade de aplicações (sistemas embarcados, sistemas de *software* e *hardware*).
- Não é afetado pela probabilidade de ocorrência de um erro.
- Produz contra-exemplos.
- Pode ser integrada no ciclo de desenvolvimento de *software*.
- Possui bases matemáticas sólidas.

Desvantagens:

- Provas podem ajudar a entender melhor um programa.
- Especificações em lógica temporal podem ficar muito complexas.
- É difícil escrever especificações.
- Não é apropriada para aplicações que manipulam muitos dados.
- Verifica um modelo do sistema e não o sistema em si. Logo exige muito na etapa de modelagem.
- Não há garantia de completude.
- O problema da explosão de estados.

3.6 FERRAMENTAS

Existem inúmeras ferramentas de *model checking*. A maioria delas inclui uma linguagem de modelagem para representar o programa, uma linguagem lógica para representar propriedades e um algoritmo de *model checking* para fazer a verificação do modelo (EMERSON,

2008). A seguir são descritas algumas das principais:

SPIN: é um sistema de verificação genérico voltado para a verificação de processos assíncronos. Utiliza uma linguagem de modelagem própria, a Promela, e fórmulas da LTL para a verificação de propriedades (HOLZMANN, 1997);

JavaPathfinder: utilizado para *model checking* de programas Java[3]. Inicialmente, era uma ferramenta que convertia código Java para Promela (HAVELUND, 1999);

NuSMV: *model checker* simbólico que utiliza CTL e LTL para a descrição e verificação de sistemas (CIMATTI *et al.*, 2002);

UPPAAL: ferramenta de verificação de sistemas de tempo real. Utiliza o autômato temporal como linguagem de modelagem e a TCTL como linguagem de propriedades (UPPAAL, 2014);

PRISM: *model checker* para a modelagem de sistemas que possuem comportamento probabilístico. Possui uma linguagem própria para a descrição de sistemas e utiliza várias lógicas temporais para a verificação de propriedades (KWIATKOWSKA; NORMAN; PARKER, 2011).

Das ferramentas listadas acima, a escolhida para a realização deste trabalho foi o UPPAAL pela possibilidade do uso de autômatos temporais, que são mais indicados para modelar os intervalos de tempo utilizados em um protocolo de controle de acesso ao meio.

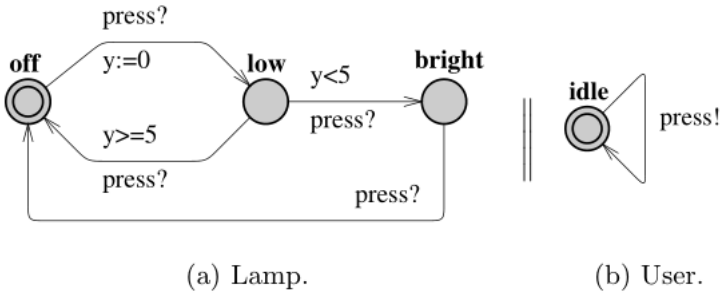
3.7 UPPAAL

UPPAAL (UPPAAL, 2014) é uma ferramenta desenvolvida em conjunto pelas universidades de Uppsala, na Suécia e Aalborg na Dinamarca (BEHRMANN; DAVID; LARSEN, 2004).

Um sistema é modelado no UPPAAL como uma rede de autômatos temporais que rodam em paralelo (*processos*). Cada transição de aresta de um autômato pode fazê-lo individualmente ou em sincronia com outro processo. Um estado do autômato, é chamado *location* e o estado de um **sistema** é definido pelas *locations* de todos os autômatos, dos valores de seus relógios e outras variáveis discretas que eles possuam (BEHRMANN; DAVID; LARSEN, 2004). Os autômatos temporais do UPPAAL podem, ainda, ser complementados com uma seção de declarações que possui uma sintaxe próxima à da linguagem C[4] de forma a facilitar a modelagem de alguns aspectos do programa como laços de repetição e estruturas de dados.

[3] (ARNOLD; GOSLING; HOLMES, 2000)
[4] (RITCHIE, 1988)

A Figura 6 a seguir mostra um sistema com dois processos modelados na ferramenta. Um processo define o funcionamento de uma lâmpada, com as posições desligada (off), fraca (low) e forte (bright). O outro processo modela um botão. A variável y é um relógio que é reiniciado na transição de off para low e testado nas transições de low para bright e low para off. As transições que têm rótulos **press!** e **press?** são canais de sincronização, ou seja, acontecem de maneira sincronizada entre os dois processos (botão e lâmpada). Note que no UPPAAL, o tempo sempre está correndo, *i.e.* a variável y está sempre sendo incrementada.



(a) Lamp. (b) User.

Figura 6 – Exemplo de processo UPPAAL

Fonte: Behrmann, David e Larsen (2004, pg. 2)

O funcionamento do sistema é o seguinte. A cada vez que o botão é "pressionado", a transição do autômato **b** é tomada e como ela contém um canal de sincronização ativo (**!**) todas as *locations* de todos os outros autômatos do sistema (neste caso, há apenas o autômato **a**) que tiverem possibilidade de tomar uma transição sobre **?** - desde que permitido pelos testes ($y < 5$, $y >= 5$) - o farão. Desta forma, ao primeiro pressionamento do botão, o autômato **a** sairá da *location* off e irá para low. Ao segundo pressionamento do botão, se menos de 5 unidades de tempo tiverem se passado, o autômato **a** irá para bright, caso contrário, para off novamente.

O exemplo apresentado acima, ilustra o processo de simulação do UPPAAL e provê uma noção do tipo de interações entre processos que podem ser modelados utilizando-se a ferramenta.

A linguagem de definição de propriedades do UPPAAL é uma versão simplificada da TCTL e consiste de fórmulas para estados e fórmulas para caminhos (BEHRMANN; DAVID; LARSEN, 2004). As fórmulas para caminhos se referem às propriedades de alcance, segurança e progresso (apresentadas na Seção 3.3), enquanto que as fórmulas de estado servem para testar se um processo está em determinado estado da seguinte forma: Processo.estado.

3.7.1 A interface do UPPAAL

O UPPAAL possui uma interface gráfica que auxilia na construção e verificação de modelos. A Figura 7 mostra a visão da aba de edição da ferramenta. Através desta aba, é possível modelar um autômato e criar e atribuir valores a variáveis de controle.

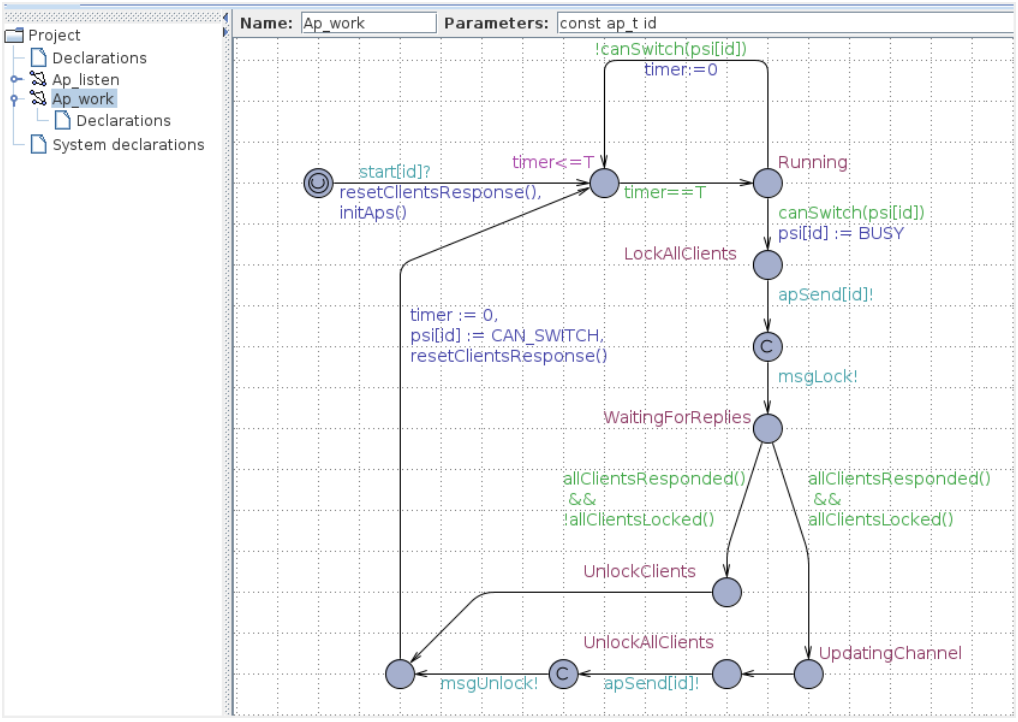


Figura 7 – Editor do UPPAAL

Fonte: Autoria própria

Após feita a edição do modelo, a ferramenta permite a verificação através de uma interface com o verificador (Figura 8). Caso a propriedade verificada não seja satisfeita, a aba de simulação apresentará o caminho que levou ao estado inválido.

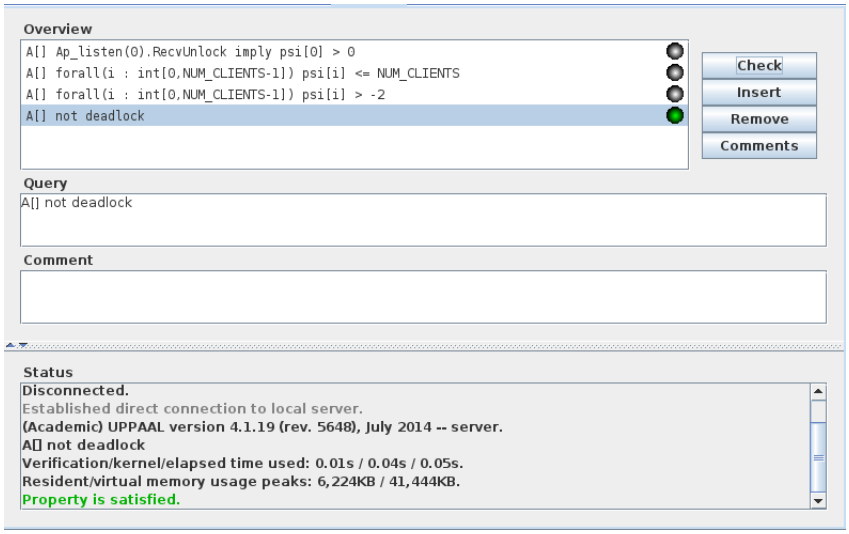


Figura 8 – Verificador do UPPAAL

Fonte: Autoria própria

Na aba de simulação (Figura 9) é possível percorrer o autômato para verificar a consistência do modelo. É possível também checar em que ponto do sistema ocorreu um erro, caso o verificador tenha gerado um caminho.

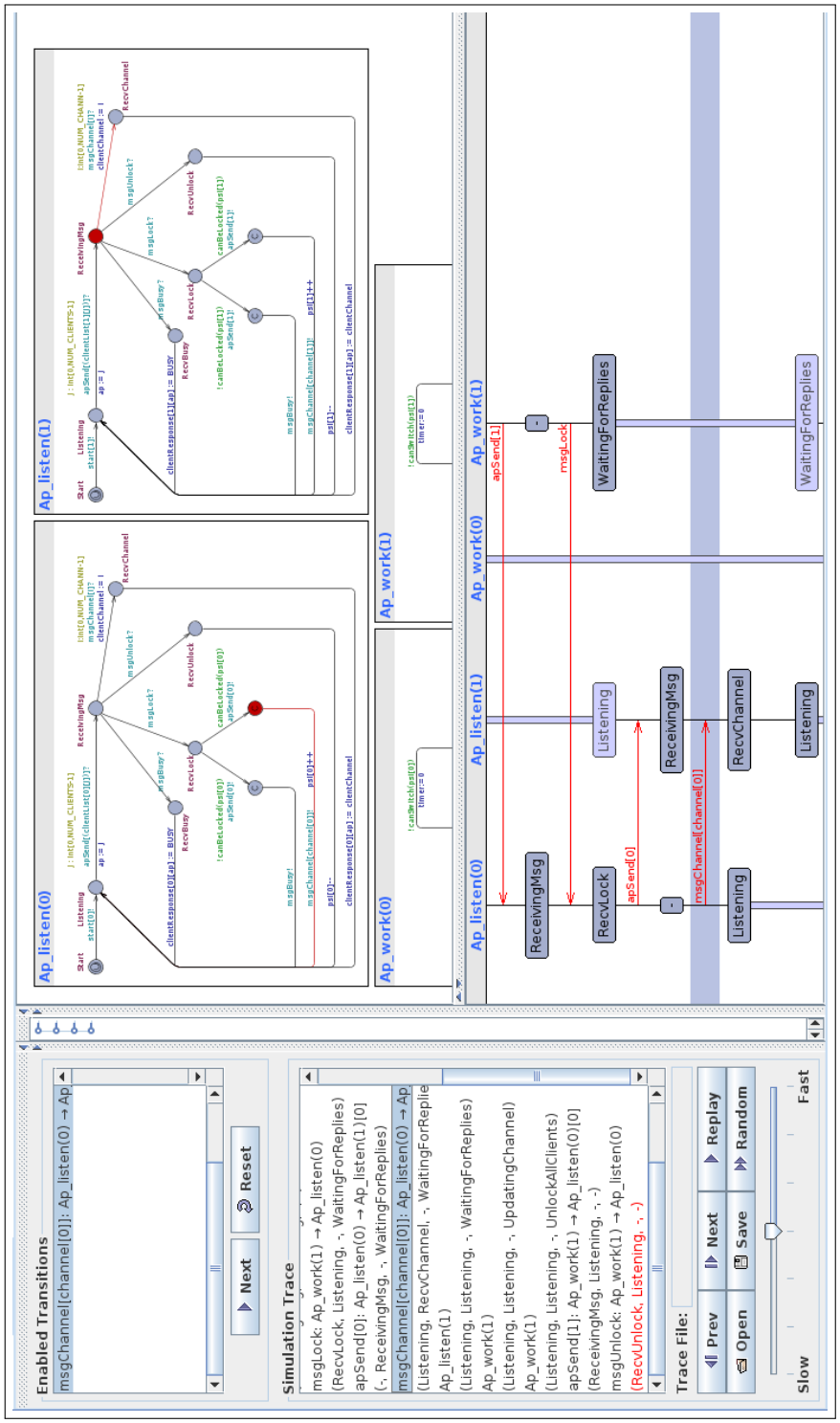


Figura 9 – Simulador do UPPAAL

Fonte: Autoria própria

3.7.2 Padrões de Modelagem

Em (BEHRMANN; DAVID; LARSEN, 2004), são descritos alguns padrões de modelagem para serem usados na ferramenta. O objetivo destes padrões é facilitar a modelagem com a inserção de estruturas padrão. Alguns padrões também têm o objetivo de diminuir a complexidade do modelo.

3.7.2.1 Redução de variáveis

A redução de variáveis consiste em reiniciar o valor de variáveis inativas. Uma variável v é considerada inativa em uma *location* l se em todos os caminhos que se iniciam em l, v será reiniciada antes de ser utilizada. Neste caso, todas as arestas que chegam em l devem conter uma ação reiniciando v.

A aplicação deste padrão garante que estados que diferem apenas no valor de uma variável inativa sejam considerados como o mesmo estado, reduzindo, assim o tamanho do espaço de estados.

No caso de relógios, este padrão é aplicado automaticamente pelo UPPAAL em um processo denominado redução ativa de relógios (*active clock reduction*).

3.7.2.2 Passagem de valor síncrona

Na ausência de estruturas explícitas para a passagem de valores entre processos, o UPPAAL se utiliza de duas estratégias para a passagem de valores. A primeira consiste em utilizar uma variável global que é escrita pelo processo que envia e lida pelo processo que recebe os dados. Isto pode ser feito de maneira unidirecional ou bidirecional e de forma condicional ou incondicional.

Passagem de valor unidirecional: O processo que envia irá prover uma sincronização (**!**) e escrever na variável global enquanto que o processo que recebe irá possuir uma sincronização (**?**) e irá ler a variável global;

Passagem de valor bidirecional: A passagem bidirecional funciona em duas etapas. Na primeira etapa, cada processo escreve em uma variável global que na segunda etapa será lida pelo outro processo. A *location* intermediária é marcada como *committed* para tornar a operação atômica;

Passagem de valor incondicional: A passagem de valor incondicional funciona sem nenhuma restrição no recebimento das variáveis;

Passagem de valor condicional: Na passagem de valor condicional, existe uma invariante na *location* que está recebendo os dados, fazendo com que a transição só aconteça (e consequentemente a passagem de valor), se a invariante puder ser satisfeita ao término da transição.

A Figura 10 ilustra estes quatro casos.

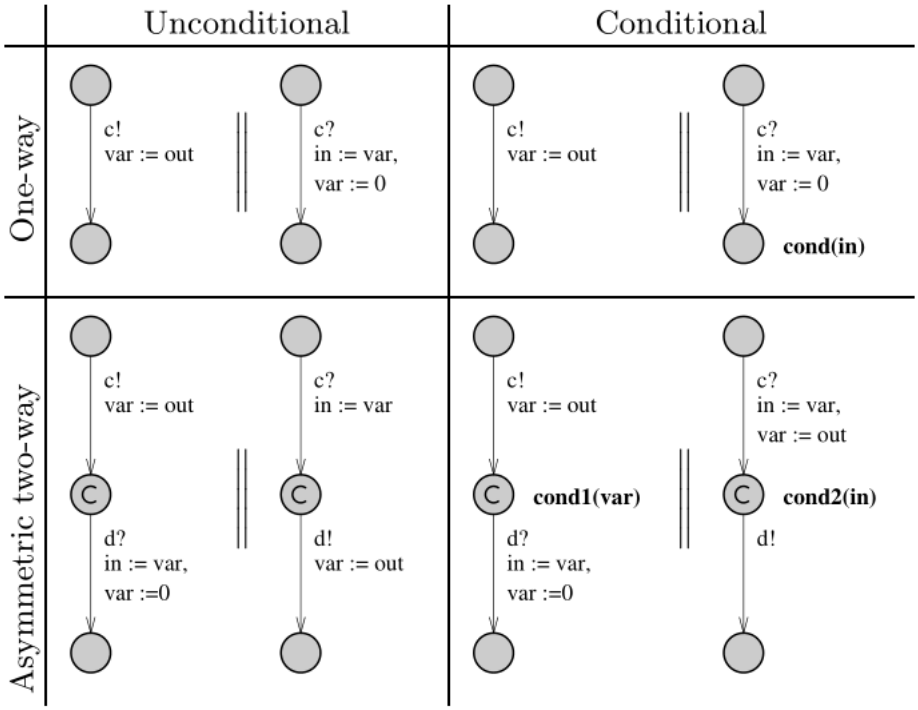


Figura 10 – Passagem de valor usando variável global

Fonte: Behrmann, David e Larsen (2004)

O outro tipo de passagem de valor não faz uso de variáveis globais. É feita uma passagem de valores implícita que serve para quando se deseja transmitir número inteiros de tamanho pequeno. A estrutura do padrão consiste em utilizar *arrays* de canais para transmitir números entre um valor mínimo e um máximo. Declara-se um *array* de canais:

```
chan send[MAX-MIN+1]
```

Para definir o valor de MAX e MIN, utiliza-se de um *select* na aresta em que está o canal da seguinte forma:

```
random:int[MIN,MAX]
send[random-MIN]!
```

O receptor fará a sincronização correspondente para receber a mensagem. A Figura 11 mostra um exemplo do uso deste padrão.

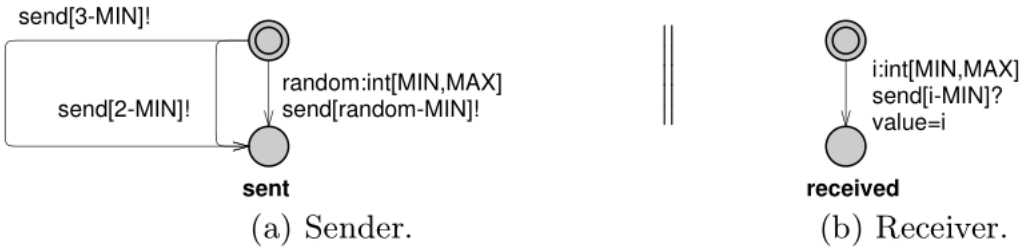


Figura 11 – *Synchronous value passing*

Fonte: Behrmann, David e Larsen (2004)

Dependendo da sincronização que for realizada, é possível determinar qual é o índice do *array* que possui aquele canal e assim saber qual é o número inteiro que está sendo transmitido.

3.7.2.3 Multicast

O UPPAAL provê canais *broadcast*, que são sincronizados entre todos os processos, porém não há como modelar a situação em que um número $\leq N$ de processos realize a sincronização. Para isto existe o padrão de *multicast*.

Para realizar uma sincronização em *multicast*, basta adicionar uma váriavel como contador do número de processos que se deseja sincronizar e utilizar um canal normal. Cada processo receptor irá então incrementar esta variável quando estiver pronto para sincronizar. No entanto, o remetente apenas está disponível para enviar quando um número maior ou igual ao número de processos que se deseja sincronizar. A Figura 12 exemplifica este padrão.

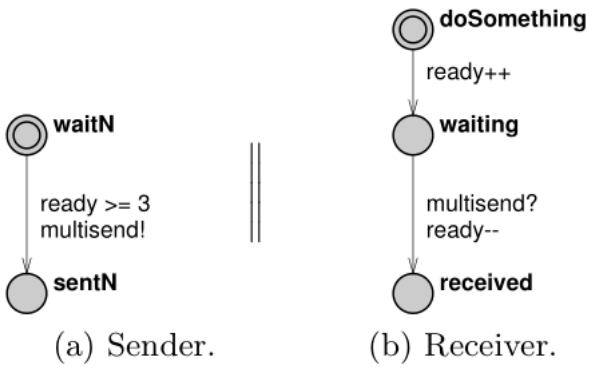


Figura 12 – Modelagem de *multicast*

Fonte: Behrmann, David e Larsen (2004)

3.7.2.4 Arestas urgentes

Pode-se dizer que um processo deve sair de uma *location* sem que o tempo passe através da utilização de *locations* urgentes. Porém não há como dizer que uma transição deve acontecer assim que possível. Em alguns casos, isto seria desejável para evitar que um processo fique mais tempo do que o mínimo necessário em uma *location*. Para representar esta situação, utiliza-se de arestas urgentes, que nada mais são do que arestas que possuem uma sincronização **?** através de um canal urgente com um outro processo que contém um laço com a sincronização **!**. Isto fará com que o primeiro processo execute esta aresta assim que possível. Veja a Figura 13.

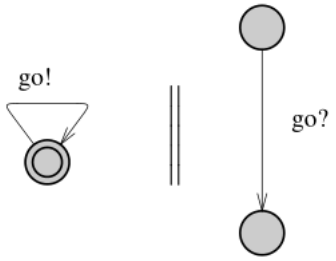


Figura 13 – Modelagem de uma aresta urgente

Fonte: Behrmann, David e Larsen (2004)

4 PROTOCOLOS DE ACESSO AO MEIO

Em WLANs, o meio de transmissão é *broadcast* e vários dispositivos podem acessá-lo ao mesmo tempo. À medida que aumenta o número de usuários na rede, aumenta a probabilidade de ocorrer mais de uma transmissão simultânea, ocasionando colisão. Por esse motivo, um esquema de controle de acesso ao meio (MAC) deve ser utilizado (GUMMALLA; LIMB, 2000). O padrão 802.11 estabelece dois mecanismos básicos de acesso, a *distributed coordination function* (DCF) e a *point coordination function* (PCF) (XU; SAADAWI, 2001). A seguir, descreve-se o funcionamento da função DCF. A função PCF não será aqui descrita por encontrar-se fora do escopo deste trabalho.

4.1 *DISTRIBUTED COORDINATION FUNCTION*

O principal mecanismo de acesso do padrão IEEE 802.11 é uma função de coordenação distribuída (DCF) que emprega o *Carrier Sense Multiple Access with Collision Avoidance* - CSMA/CA (IEEE..., 2012).

O protocolo CSMA/CA foi projetado para reduzir a probabilidade de colisão entre múltiplas estações. Apenas esperar que o meio se torne livre para então transmitir não é suficiente para evitar colisões por que múltiplas estações podem estar aguardando o meio se tornar livre. Quando o meio eventualmente se tornar livre, todas as estações irão percebê-lo como livre ao mesmo tempo, e transmitirão simultaneamente, ocasionando colisão (IEEE..., 2012).

A coordenação das estações é feita através do uso de períodos de tempo definidos com base na camada física mais independentemente das taxas de transmissão disponíveis. O uso destes intervalos permite que se evite a colisão de transmissões e garante que *frames* com maior prioridade tenham que esperar menos tempo para serem transmitidos. Os principais espaços *interframe* que são utilizados no 802.11 são (GAST, 2005):

SIFS (*short interframe space*): é utilizado para transmissões de prioridade, como por exemplo um *frame* de confirmação que é enviado após o recebimento de um *frame* de dados;

DIFS (*DCF interframe space*): é o período mínimo de tempo que as estações precisam esperar antes de transmitir;

EIFS (*extended interframe space*): período de tempo extendido. Usado quando a estação recebe um frame corrompido.

Se uma estação deseja transmitir, ela deve primeiro sensoriar o meio para determinar se outra estação já está transmitindo. Se o meio estiver livre por um período de DIFS, a transmissão

pode iniciar. Este período é necessário para que exista um tempo mínimo entre a transmissão de dois *frames* consecutivos (IEEE..., 2012).

Caso seja determinado que o meio está ocupado, a estação deverá aguardar até que a transmissão em andamento termine. Após isso, ela deverá selecionar um número aleatório de espera (chamado contador de *backoff*) a partir de um intervalo de tempo chamado janela de disputa (*contention window*). Em seguida, a estação contabilizará um tempo de espera com base no valor de *backoff* sorteado para poder transmitir um *frame* de dados. O valor do contador de *backoff* será decrementado enquanto o meio encontrar-se livre e quando o seu valor chegar a zero, a estação poderá então transmitir. Este processo é também efetuado logo após uma transmissão efetuada com sucesso (IEEE..., 2012). O procedimento de *backoff* será detalhado a seguir.

4.1.1 Procedimento de *Backoff*

O processo de *backoff* é realizado toda vez que uma estação detecta o meio livre por DIFS ou EIFS, conforme apropriado. A estação deverá sortear um número para o seu contador de *backoff* através da equação:

$$\text{BackoffTime} = \text{Random}() \times \text{aSlotTime}, \text{ onde:}$$

Random() = número inteiro selecionado do intervalo [0,CW] de acordo com uma distribuição uniforme, onde CW é um inteiro tal que $\mathsf{aCWmin} \leq \mathsf{CW} \leq \mathsf{aCWmax}$ sendo *aCWmin* e *aCWmax* constantes definidas pela camada física sendo utilizada.

aSlotTime = valor da duração de um *slot* em uma unidade de tempo absoluta (*e.g.* microssegundos). Corresponde ao atraso para sensoriar o meio, ao tempo para trocar de modo de recepção para modo de transmissão, ao tempo de propagação da onda e ao tempo de resposta da camada MAC acumulados.

Caso o contador de *backoff* já contenha um número diferente de zero, o procedimento acima não é realizado e a contagem continua de onde estava.

Se nenhuma atividade for detectada no meio pela duração de um *slot*, o contador de *backoff* é decrementado de um $aSlotTime$. Caso o meio seja detectado como ocupado durante qualquer momento deste período, o processo de *backoff* é suspenso. Para que a contagem de *backoff* possa ser retomada, o meio deverá ser detectado como livre por um período de DIFS, ou EIFS, conforme visto anteriormente. Quando o contador de *backoff* atinge o valor zero, a transmissão se inicia (IEEE..., 2012).

A Figura 20 a seguir ilustra o processo de *backoff*. Inicialmente, o meio encontra-se livre e após a passagem de DIFS (1), uma estação começa a transmitir, deixando o meio ocupado (2). Enquanto isso, todas as outras estações irão esperar até que o meio fique livre por DIFS (3). Após o término da transmissão, a estação destino irá esperar SIFS (4) e enviar o seu ACK (não

representado) enquanto que as outras estações irão ouvir o meio por DIFS (4). Quando o meio tornar-se livre por DIFS, todas as estações que estavam aguardando irão iniciar (ou retomar) o processo de *backoff* (5) e decrementar o valor do contador de *backoff* enquanto o meio estiver livre (6). Quando o contador de *backoff* de uma das estações chegar a zero, a transmissão de um *frame* irá ocorrer (7).

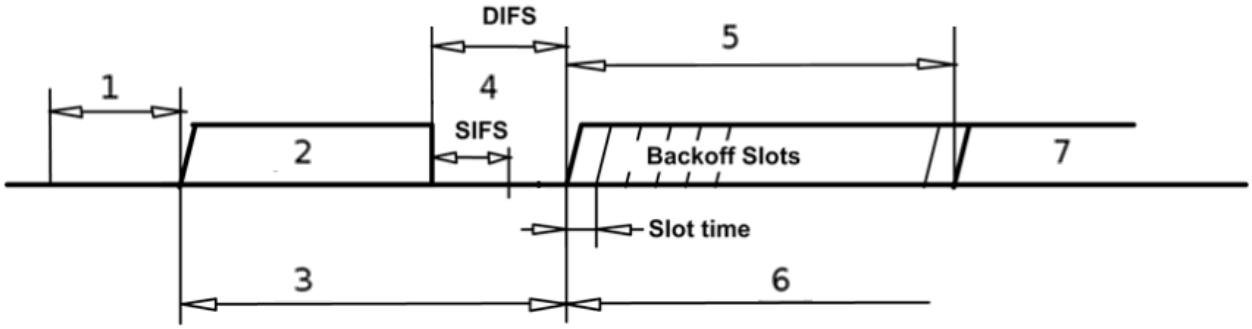


Figura 14 – O processo de backoff

Fonte: Adaptado de IEEE... (2012, pg. 826)

4.1.2 Acknowledgement

O padrão define que a estação que está recebendo os *frames* deve responder com uma confirmação (*acknowledgement*) se o *frame* for recebido corretamente. A ausência de recepção de um ACK pela estação que iniciou a comunicação indica que algum erro ocorreu durante a transmissão do *frame* inicial ou durante a transmissão do ACK, embora não haja como a estação diferenciar os dois casos (IEEE..., 2012).

A transmissão de um ACK deve ser precedida pela espera de um período de SIFS, enquanto que a estação que está esperando o ACK irá aguardar por um período de ACKTimeout = SIFS + aSlotTime + aPHY-RX-START-DELAY para recebê-lo[1] (IEEE..., 2012). Caso um outro *frame* que não o ACK seja detectado sendo transmitido, ou caso o ACK não chegue dentro de ACKTimeout, a estação irá presumir que sua transmissão não foi bem sucedida e irá pleitear o reenvio do *frame* (ZIOUVA; ANTONAKOPOULOS, 2002).

Na Figura 15 está representada a comunicação entre duas estações com uma terceira estação presente. Observe que após a transmissão de um *frame* de dados, a estação de destino irá aguardar por SIFS e então transmitir o ACK. A estação que está de fora da transação irá aguardar até o meio se tornar livre por DIFS e iniciar o processo de backoff.

[1] aPHY-RX-START-DELAY é o tempo necessário para ler o preâmbulo do cabeçalho do frame

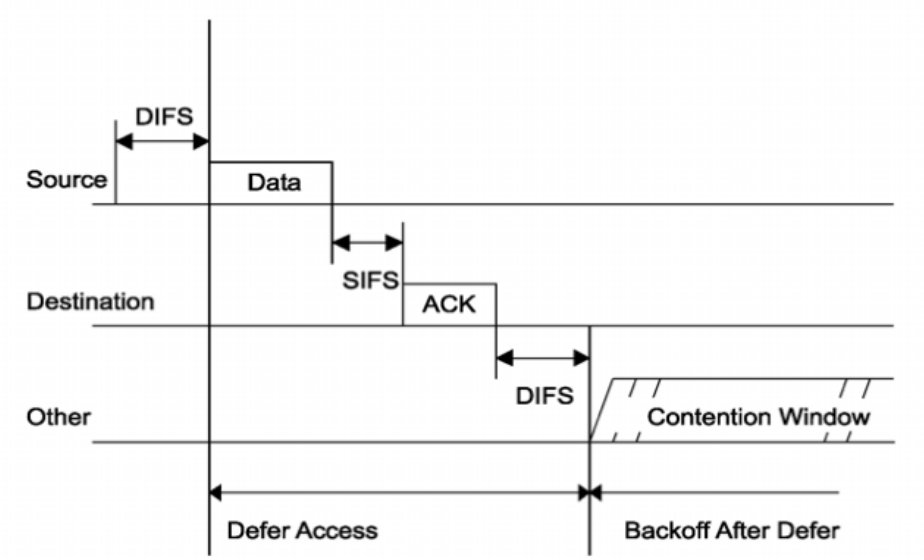


Figura 15 – Transmissão de um frame de dados e de um ACK

Fonte: Adaptado de IEEE… (2012)

Quando ocorre uma transmissão com sucesso - o que é marcado pela recepção de um ACK - a estação que iniciou a comunicação irá reverter o valor de CW para *aCWmin* e passar por um novo período de backoff antes de enviar seu próximo frame. Caso a transmissão não seja realizada com sucesso, a estação irá incrementar o valor de CW da seguinte forma: $CW = ((CW + 1) * 2) - 1$ até o máximo de *aCWmax*. A Figura 16 mostra o processo de expansão da janela de contenção a cada transmissão mal sucedida. Quando a janela de contenção atinge seu valor máximo, ela será repetida até que a transmissão seja realizada com sucesso (IEEE…, 2012).

O mecanismo citado acima é o mecanismo básico de funcionamento do controle de acesso ao meio do padrão IEEE 802.11. Existe ainda um refinamento deste método que tem como objetivo minimizar ainda mais as colisões, incluindo pequenos frames de controle (*Request to send* - RTS e *Clear to send* - CTS) que são trocados pelas estações envolvidas na comunicação, porém sob a pena de um maior *overhead* na comunicação. Este método não será tratado neste documento.

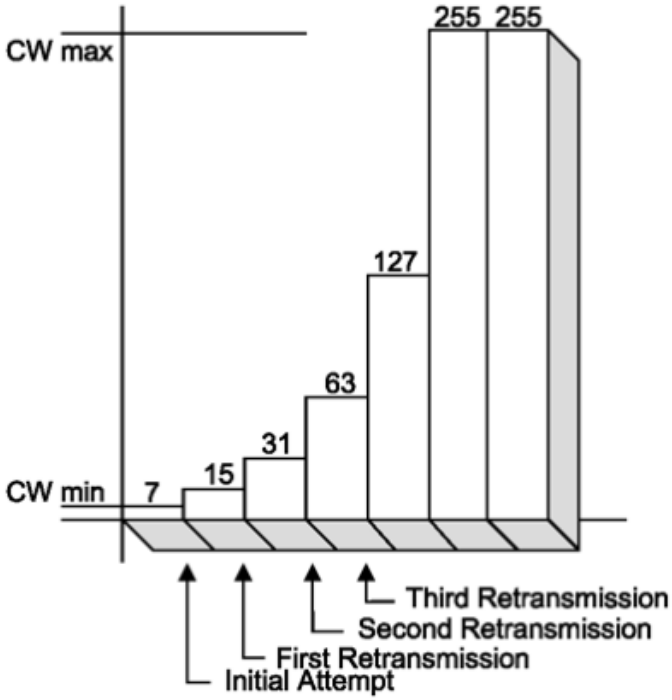


Figura 16 – Expansão da janela de contenção

Fonte: Adaptado de IEEE… (2012)

5 DESENVOLVIMENTO

Este capítulo descreve a metodologia e as etapas de realização deste trabalho. Na Seção 5.1 apresenta-se as suposições feitas com relação à área de redes. Em 5.2 descreve-se a modelagem realizada e faz-se uma correspondência entre as etapas do processo de acesso ao meio e o modelo formal criado. Na Seção 5.3 lista-se as propriedades que foram verificadas e os resultados obtidos.

5.1 METODOLOGIA

A modelagem realizada foi feita com base no documento do padrão IEEE 802.11 (IEEE..., 2012). Modelou-se o método de acesso básico (*Basic Access* - BA) conforme a seção 9.3.4.2 do padrão. Supõe-se um modelo saturado com o mesmo tempo de transmissão para todos os nós. No modelo, todos os nós iniciam o seu funcionamento simultaneamente. Supõe-se, ainda, que o meio é perfeito e nenhuma mensagem é perdida, bem como a ausência de fontes de interferência externas.

Para os valores dos intervalos de tempo, utilizou-se de um pacote GNU Octave (EATON; BATEMAN; HAUBERG, 2009) que efetua o cálculo de vazão do protocolo. Foi escrito um *shell script* que executa este pacote, extrai os dados de interesse e edita o arquivo do modelo, inserindo os valores na forma de declarações de variáveis.

Para a definição das propriedades a serem verificadas, definiu-se um tempo ideal de base para a transmissão dos n nós que foi usado como referência na construção das fórmulas lógicas conforme exposto na Seção 5.3.

5.2 MODELAGEM

O modelo criado representa uma estação que pretende transmitir uma sequência infinita de *frames* e para tanto deve passar pelo processo de disputa do meio. Ao vencer a disputa, a estação irá enviar seu *frame* a uma outra estação, escolhida de forma não-determinística, conforme disposto na Seção 5.2.1 a seguir. Ao receber um *frame*, cada estação enviará um ACK para a estação que iniciou a comunicação.

Como o UPPAAL não possui meios explícitos para a troca de mensagens entre autômatos, utilizou-se do padrão de modelagem de passagem síncrona de valor, citado na Seção 3.7.2.2, para simular uma situação de troca de mensagens.

5.2.1 Troca de Mensagens

A idéia principal por trás da troca de mensagens é que números inteiros de tamanho pequeno sejam passados de um processo a outro. Estes números simbolizam as *ids* dos processos e servem como uma forma de endereçamento. Utilizou-se o padrão de modelagem de passagem síncrona de valores da seguinte maneira.

Foram modelados dois canais *broadcast* para indicar a transferência de uma mensagem:

```
broadcast chan start_sending;
broadcast chan finish_sending[id_t][id_t];
```

O canal *start_sending* modela o início da transmissão. A *location* à qual este canal leva irá modelar a duração da transmissão do *frame* e o canal *finish_sending* irá modelar o término da transmissão. É no término da transmissão que está aplicado o padrão de troca de mensagens.

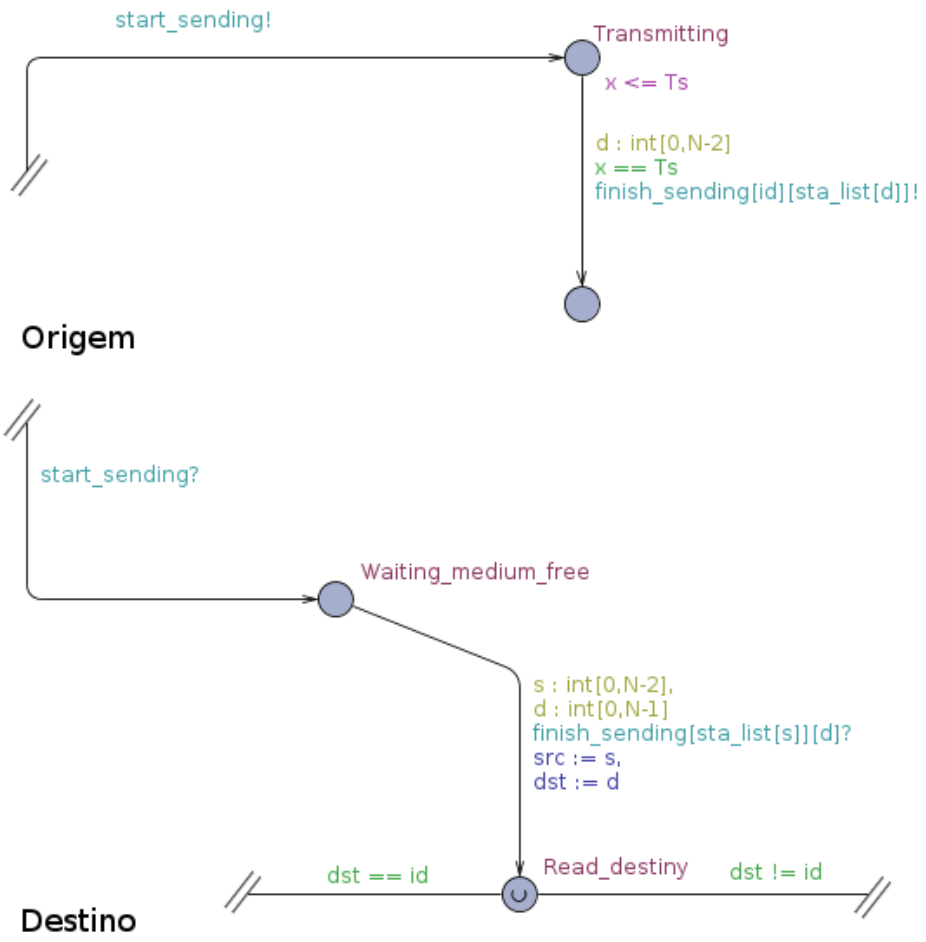


Figura 17 – Troca de mensagens

Fonte: Autoria própria

Na Figura 17, observe que após a sincronização com *start_sending*, o nó de origem irá permanecer transmitindo por um período igual a *Ts* e então irá sortear um número através do *select* da aresta e disponibilizar a sincronização de término na forma *finish_sending[origem][destino]*, onde *destino* será definido com base em uma lista de estações disponíveis (*sta_list*)[1]. O nó destino irá efetuar as ações análogas para receber a mensagem.

Após o recebimento do *frame*, a estação de destino verifica se ela é realmente o destinatário daquela mensagem comparando o valor do número que foi "recebido" em *dst* com a sua *id*.

5.2.2 *Acknowledgement*

Ao enviar um *frame* de dados, a estação deve ficar esperando por *ACKTimeout* pelo recebimento de um ACK. Neste momento, o meio pode se tornar ocupado, o que fará com que a estação vá para o estado *Waiting_medium_free*. A partir deste estado, conforme descrito anteriormente, a estação irá detectar se o *frame* que está sendo transmitido é endereçado a ela ou não. Caso afirmativo e caso o *frame* seja um ACK, a estação irá retornar o valor de *CW* para *aCWmin* e prosseguir com o processo de *backoff*, já que a transmissão foi efetuada com sucesso. Caso o ACK não seja detectado dentro de *ACKTimeout* ou seja detectada uma transmissão de um *frame* de dados ou ainda uma transmissão endereçada a outro nó, a estação irá iniciar o processo de *backoff*, mas desta vez porque a transmissão não obteve sucesso. A Figura 18 ilustra o processo.

Do ponto de vista da estação que está recebendo o *frame* de dados, será necessário aguardar por SIFS antes de poder enviar o ACK. Após o envio do ACK, a estação irá passar pelo *backoff* para disputar pelo meio. A Figura 19 a seguir mostra esse processo.

Em ambos os casos (espera pelo ACK e envio do ACK), o autômato passará pela *location Read_destiny* para determinar se o que está ocupando o meio é um *frame* endereçado a si. Esta *location* é do tipo urgente para que esta operação ocorra sem atraso de forma a melhor modelar a situação real, em que o próprio recebimento da transmissão implica em saber a quem ela foi endereçada.

[1] Isto serve para evitar que um nó sorteie o próprio *id* como destino da mensagem

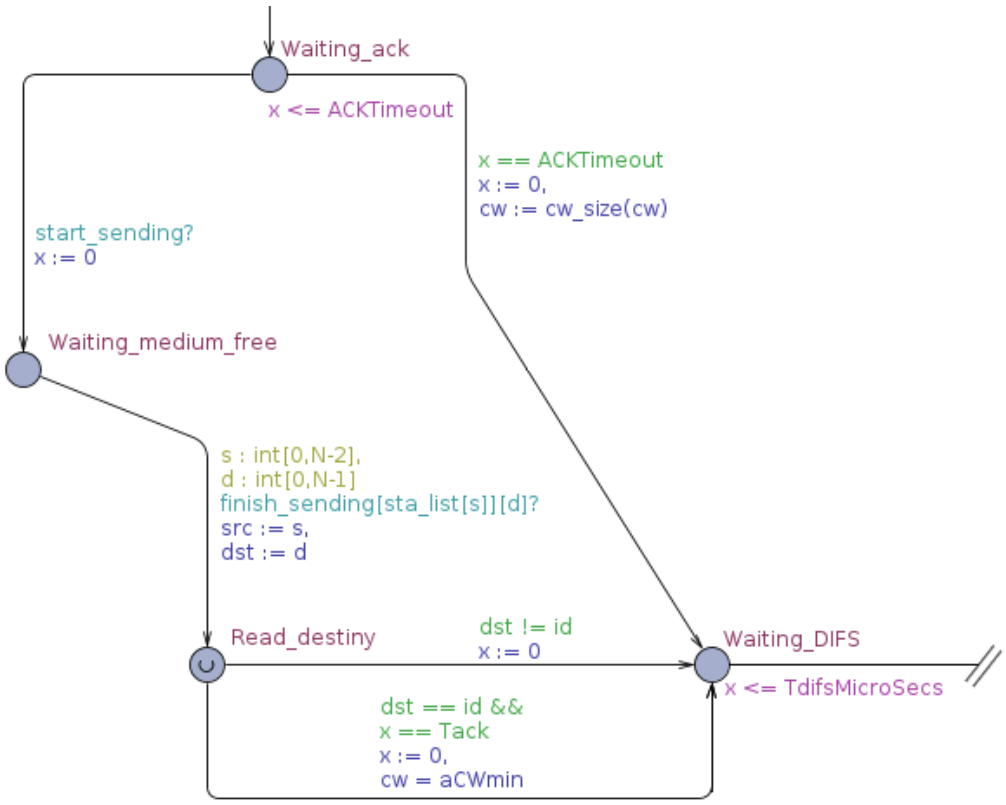


Figura 18 – Espera pelo ACK

Fonte: Autoria própria

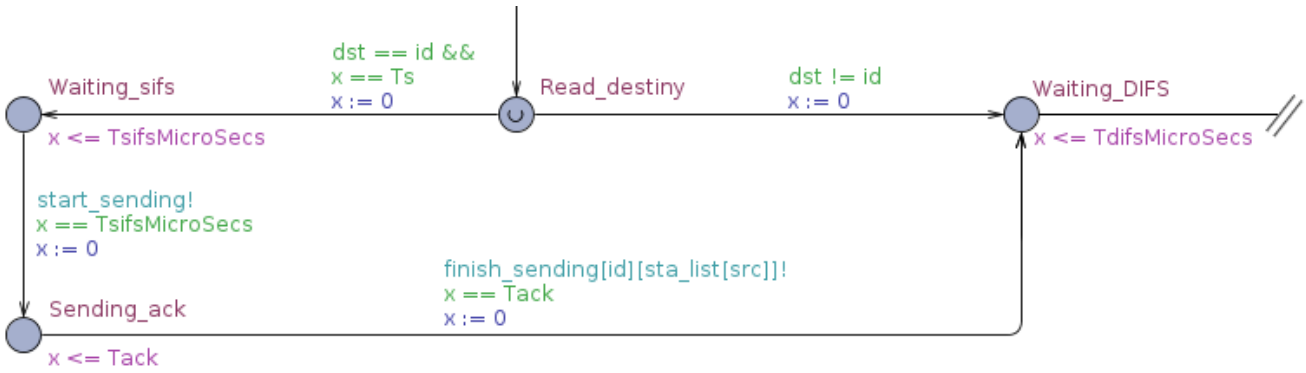


Figura 19 – Envio do ACK

Fonte: Autoria própria

5.2.3 *Backoff*

O processo de *backoff* foi modelado como mostra a Figura 20. De baixo para cima, a *location Waiting_DIFS* representa o momento em que a estação detecta o meio livre por DIFS e deverá sortear um número de *backoff*. A *location Waiting_DIFS* possui duas arestas de saída para suportar o fato de que o processo de *backoff* pode estar se iniciando agora, ou pode estar sendo retomado devido a uma interrupção anterior.

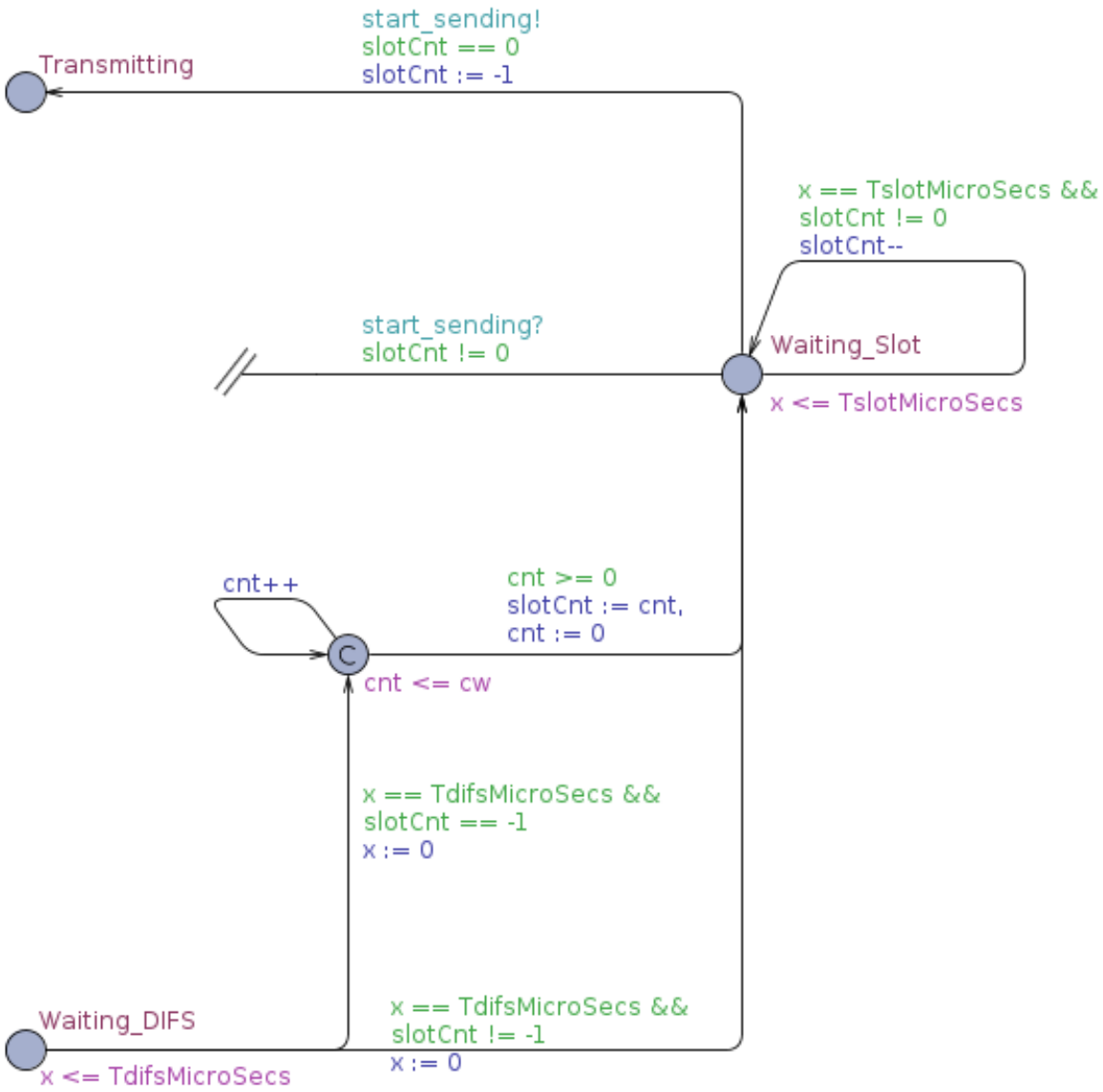


Figura 20 – Modelagem do *backoff*

Fonte: Autoria própria

A próxima *location* é responsável por simular um gerador de números aleatórios. Como está marcada como *commited*, o sorteio de um número ocorre sem intercalação de outros pro-

cesso, fazendo com esta operação seja atômica. Existem duas arestas de saída, uma que possui a ação de incrementar o contador *cnt* e outra que transfere o valor de *cnt* para *slotCnt* e prossegue com o processo de *backoff*. A geração de um número aleatório se dá pelo fato de que, como as arestas têm a mesma probabilidade de ocorrer, *i.e.* não há nada que indique preferência de uma aresta sobre a outra, uma das duas será escolhida não-deterministicamente fazendo com que o valor de *cnt*++ seja aleatório.

Quando a *location Waiting_Slot* for atingida, o autômato permanecerá nesta *location* pelo tempo de *slot* $\times$ *slotCnt* ou até que o meio se torne ocupado, caso no qual a aresta *start_sending?* ocasionará a interrupção do processo de *backoff*. Caso *slotCnt* chegue a zero, a transmissão se inicia.

Além das etapas mencionadas aqui (envio e recebimento de mensagens e *acknowledgement* e *backoff*), o modelo contém ainda uma chamada para a função *initNetwork*[2] no estado inicial e arestas que levam cada uma das *locations*, exceto *Transmitting* e *Waiting_sifs*[3], à *location Waiting_medium_free*. O modelo completo está disposto na Figura 21, na página seguinte.

[2] função que inicia o *array sta_list* com os índices de todos os nós, exceto o atual.
[3] estas são operações realizadas independentemente do estado do meio.

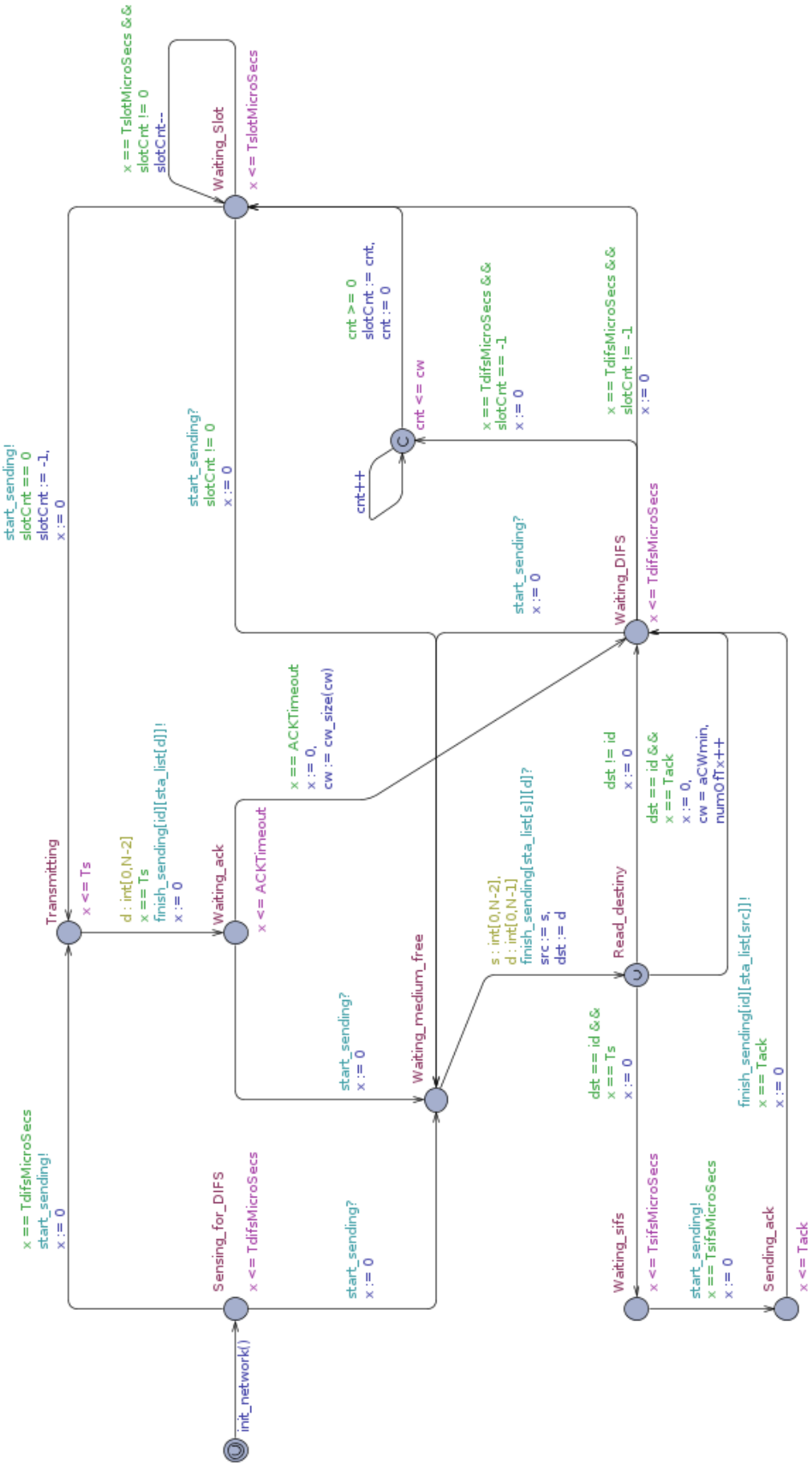


Figura 21 – Modelo completo

Fonte: Autoria própria

5.3 VERIFICAÇÃO

Primeiramente, é necessário garantir a ausência de *deadlock* no modelo. Isto é feito através da fórmula $\forall\Box\neg deadlock$ e, no verificador do UPPAAL $A[]\ not\ deadlock$. A verificação desta propriedade mostra que não há *deadlocks* para $n = 2$. Para $n = 3$, não foi possível efetuar a verificação devido à explosão do espaço de estados[4].

Para estabelecer um parâmetro de comparação para a verificação dos nós, é possível determinar o *tempo ideal teórico* para que todos os nós transmitam com sucesso como sendo:

$$n \times Ts, \text{ onde:}$$

- n é o número de nós;
- Ts é o tempo mínimo necessário para efetuar uma transmissão com sucesso, onde sucesso é definido como uma transmissão sucedida do recebimento de um ACK.

Para este tempo ideal ser atingido em uma rede, seria necessário que cada estação iniciasse seu funcionamento imediatamente após o término da transmissão da estação anterior, formando uma fila. Este tempo, contudo, não é alcançavel, já que nas suposições feitas neste trabalho, os nós iniciam sua execução simultaneamente o que faz com que precisem passar pelo processo de *backoff*. Desta forma, define-se o *tempo ideal real* como:

$$\begin{aligned}
T_1 &= 0aSlotTime + Ts \\
T_2 &= T_1 + (1aSlotTime + Ts) \\
T_3 &= T_2 + (2aSlotTime + Ts) \\
T_4 &= T_3 + (3aSlotTime + Ts) \\
&\vdots \\
T_n &= T_{n-1} + ((n-1)aSlotTime + Ts)
\end{aligned}$$

O apêndice A contém uma prova por indução da generalização acima. Resolvendo a equação de recorrência, temos:

$$T_n = \sum_{i=0}^{n-1} iaSlotTime + nTs$$

Logo:

$$T_n = \frac{n(n-1)}{2} aSlotTime + nTs$$

Este tempo é obtido levando-se em consideração que cada estação terá de esperar até que todas as estações anteriores terminem suas transmissões para evitar colisões. Desta forma, o tempo mínimo de espera entre duas transmissões é o tempo levado pela estação anterior adicionado ao tempo de transmissão da estação atual mais um *slot* de tempo para garantir a ordenação das transmissões.

[4] Ver a seção de trabalhos futuros (6)

A avaliação das propriedades será feita com base neste tempo. A partir deste valor, podemos fazer as seguintes perguntas:

1. é possível um nó jamais transmitir dentro de um intervalo de tempo Tn?
   Esta pergunta corresponde a verificar se há *starvation* no sistema e pode ser expressa pela fórmula temporal:

   $$\exists i : i \in \mathbb{N} \wedge i \in [0, N-1] \ / \ \exists \Diamond (y \geq Tn \wedge \neg transmitir_i)$$

   Na linguagem do verificador do UPPAAL, a propriedade fica:

   $$E <> exists(i : int[0, N-1])(y >= Tn \ \&\& \ Node(i).numOfTx == 0)$$

   onde ***numOfTx*** é uma variável que é incrementada a cada transmissão efetuada com sucesso.

   A verificação desta propriedade mostrou ser possível que um nó nunca chegue a transmitir. No caso de $n = 2$, é possível que um dos nós transmita 0 vezes e o outro transmita 2 vezes. Esta propriedade também é válida para valores de n até 4. Valores maiores que 4 não puderam ser verificados.

2. é possível um mesmo nó transmitir k vezes ($2 \leq k \leq n$) dentro do intervalo Tn? Esta pergunta representa a verificação da justiça do modelo, já que caso uma estação transmita mais de uma vez dentro de Tn, outras ficarão sem transmitir. Isto pode ser representado pela fórmula:

   $$\exists i : i \in \mathbb{N} \wedge i \in [0, N-1] \ / \ \exists \Diamond (y \leq Tn \wedge transmitir_i^{\geq 2})$$

   e no UPPAAL:

   $E <> exists(i : int[0, N-1])(y <= Tn \ \&\& \ Node(i).numOfTx >= 2 \ \&\& \ Node(i).numOfTx <= N)$

   Para $n = 2$ esta resposta é obtida através do resultado da pergunta anterior. Nesta situação é possível que um dos nós transmita 2 vezes. Para $n \geq 3$, não foi possível efetuar a verificação devido à explosão do espaço de estados.

3. é impossível não haver ao menos uma transmissão dentro de Tn? Esta propriedade representa uma verificação de segurança. O fato de não haver transmissões dentro de Tn é algo indesejável. Em lógica temporal:

   $$\forall i : i \in \mathbb{N} \wedge i \in [0, N-1] \ / \ \exists \Diamond (y \geq Tn \wedge \neg transmitir_i)$$

   E na linguagem do verificador:

$$E <> forall(i : int[0, N-1])(y >= Tn \;\&\&\; Node(i).numOfTx == 0)$$

Esta verificação não pode ser efetuada devido à explosão do espaço de estados com número de nós maior ou igual a dois.

6 CONSIDERAÇÕES FINAIS

Conforme exposto no capítulo introdutório, inicialmente o objetivo deste trabalho era a verificação formal do protocolo PbP-DCF, porém devido à falta de estudos formais do protocolo CSMA/CA, optou-se por efetuar a verificação formal do mesmo que foi feita através da ferramenta UPPAAL. Alguns estudos relacionados que modelam outro padrão (802.15.4) puderam ser tomados como base para a modelagem, porém as diferenças entre o funcionamento dos dois não permitiu que nenhuma suposição fosse aproveitada dos outros modelos.

Durante a fase de modelagem, uma das principais dificuldades encontradas foi o entendimento preciso do funcionamento do padrão devido à falta de conteúdo estruturado que o representasse. Além disso, o fato de que a verificação de propriedades era em algumas vezes muito lenta, fez com que a verificação automática da modelagem fosse menos utilizada, optando-se pela execução do simulador para estes propósitos.

Um dos grandes impedimentos à etapa de verificação foi a questão da explosão de estados. Em retrospecto, fica claro que este é o principal problema na construção e verificação de um modelo formal. Aliado a isso, a lógica temporal usada pela ferramenta possui uma descrição escassa e portanto é necessário submeter algumas fórmulas ao verificador para garantir que a sua formulação esteja correta. Isto, aliado ao problema da explosão de estados faz com que o processo de verificação seja bastante demorado.

Apesar dos fatores negativos citados acima, verificou-se que o processo de criação de um modelo formal por si só já traz vantagens com relação ao entendimento de uma tecnologia, já que é necessário descrevê-la em maiores detalhes para que se possa efetuar a modelagem corretamente. Com um maior entendimento dos algoritmos de *model checking* que são utilizados pela ferramenta, seria possível estender estas vantagens ainda mais, construindo-se modelos mais eficientes e diminuindo o tamanho do espaço de estados, facilitando assim, todo o processo.

Este trabalho apresenta como principal contribuição a construção de um modelo formal para o protocolo CSMA/CA com base no documento do padrão IEEE 802.11. Este modelo poderá servir de base para trabalhos futuros que tenham como alvo de modelagem o padrão 802.11 ou mesmo outros padrões similares. Além disso, a aplicação da técnica de *model checking* e, mais especificamente, a ferramenta UPPAAL gerou um estudo que pode ser reaproveitado por futuros integrantes do grupo de pesquisa. Ainda como contribuições deste, pode-se mencionar o esforço no sentido de aplicar conhecimentos tanto da área formal quanto da área de redes na resolução de um problema, já que percebe-se que são poucos trabalhos que aplicam esta interdisciplinaridade.

Algumas oportunidades ainda restam e pedem a consideração de trabalhos futuros, tais quais:

- a verificação formal de protocolos baseados no CSMA/CA (*e.g.* PbP-DCF), tomando

como base este documento e estendendo a verificação para avaliar requerimentos e suposições específicas;

- uma investigação sobre técnicas e ferramentas para a simplificação dos modelos e diminuição do espaço de estados;
- uma avaliação do *model checking* em máquinas com maior capacidade de armazenamento e/ou *clusters* de máquinas a fim de determinar os limites práticos da técnica;
- um estudo aprofundado das tecnologias que servem de base para o UPPAAL, de modo a definir quais os melhores parâmetros para a verificação de sistemas de diversos tipos e tamanhos;

REFERÊNCIAS

ARNOLD, K.; GOSLING, J.; HOLMES, D. **The Java Programming Language, Third Edition**. [S.l.]: Addison-Wesley, 2000. ISBN 0-201-70433-1.

BAIER, C.; KATOEN, J.-P. **Principles of Model Checking (Representation and Mind Series)**. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

BEHRMANN, G. *et al.* Uppaal implementation secrets. In: DAMM, W.; OLDEROG, E.-R. (Ed.). **Formal Techniques in Real-Time and Fault-Tolerant Systems**. Springer Berlin Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2469). p. 3–22. ISBN 978-3-540-44165-6. Disponível em: <http://dx.doi.org/10.1007/3-540-45739-9_1>.

BEHRMANN, G.; DAVID, A.; LARSEN, K. G. A tutorial on UPPAAL. In: BERNARDO, M.; CORRADINI, F. (Ed.). **Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004**. [S.l.]: Springer–Verlag, 2004. (LNCS), p. 200–236.

BOCHMANN, G.; SUNSHINE, C. Formal methods in communication protocol design. **Communications, IEEE Transactions on**, v. 28, n. 4, p. 624–631, 1980. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1094685>.

BOCHMANN, G. V. Finite state description of communication protocols. **Computer Networks (1976)**, v. 2, n. 4, p. 361–372, 1978. Disponível em: <http://www.sciencedirect.com/science/article/pii/0376507578900156>.

BROWNE, M.; CLARKE, E.; GRüMBERG, O. Characterizing finite kripke structures in propositional temporal logic. **Theoretical Computer Science**, v. 59, n. 1–2, p. 115 – 131, 1988. ISSN 0304-3975. Disponível em: <http://www.sciencedirect.com/science/article/pii/0304397588900989>.

BULYCHEV, P. *et al.* Rewrite-based statistical model checking of wmtl. In: QADEER, S.; TASIRAN, S. (Ed.). **Runtime Verification**. Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7687). p. 260–275. ISBN 978-3-642-35631-5. Disponível em: <http://dx.doi.org/10.1007/978-3-642-35632-2_25>.

CHALMERS, D. *et al.* **Ubiquitous Computing: Experience, Design and Science**. [S.l.], 2006.

CIMATTI, A. *et al.* Nusmv 2: An opensource tool for symbolic model checking. In: **Proceedings of the 14th International Conference on Computer Aided Verification**. London, UK, UK: Springer-Verlag, 2002. (CAV '02), p. 359–364. ISBN 3-540-43997-8. Disponível em: <http://dl.acm.org/citation.cfm?id=647771.734431>.

CLARKE, E. M. In: GRüMBERG, O.; VEITH, H. (Ed.). **25 Years of Model Checking**. Berlin, Heidelberg: Springer-Verlag, 2008. cap. The Birth of Model Checking, p. 1–26. ISBN 978-3-540-69849-4.

CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: **Logic of Programs, Workshop**. London, UK, UK: Springer-Verlag, 1982. p. 52–71. ISBN 3-540-11212-X. Disponível em: <http://dl.acm.org/citation.cfm?id=648063.747438>.

CLARKE, E. M.; EMERSON, E. A.; SIFAKIS, J. Model checking: Algorithmic verification and debugging. **Commun. ACM**, ACM, New York, NY, USA, v. 52, n. 11, p. 74–84, nov. 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1592761.1592781>.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 28, n. 4, p. 626–643, dez. 1996. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/242223.242257>.

CLARKE JR., E. M.; GRüMBERG, O.; PELED, D. A. **Model Checking**. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-03270-8.

DIN, C. C. *et al.* Observable behavior of distributed systems: Component reasoning for concurrent objects. **J. Log. Algebr. Program.**, v. 81, n. 3, p. 227–256, 2012.

DIN, C. C.; OWE, O.; BUBEL, R. Runtime assertion checking and theorem proving for concurrent and distributed systems. In: PIRES, L. F. *et al.* (Ed.). **MODELSWARD**. [S.l.]: SciTePress, 2014. p. 480–487. ISBN 978-989-758-007-9.

EATON, J. W.; BATEMAN, D.; HAUBERG, S. **GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations**. CreateSpace Independent Publishing Platform, 2009. ISBN 1441413006. Disponível em: <http://www.gnu.org/software/octave/doc/interpreter>.

EMERSON, E. A. 25 years of model checking. In: GRüMBERG, O.; VEITH, H. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2008. cap. The Beginning of Model Checking: A Personal Perspective, p. 27–45. ISBN 978-3-540-69849-4. Disponível em: <http://dx.doi.org/10.1007/978-3-540-69850-0_2>.

FRUTH, M. **Formal Methods for the Analysis of Wireless Network Protocols**. Tese (Doutorado) — Oxford University, 2011. Disponível em: <http://www.prismmodelchecker.org/papers/matthias-fruth-phd.pdf>.

GAST, M. S. **802.11 Wireless Networks: The Definitive Guide, Second Edition**. [S.l.]: O'Reilly Media, Inc., 2005. ISBN 0596100523.

GUMMALLA, A. C. V.; LIMB, J. O. Wireless medium access control protocols. **Communications Surveys Tutorials, IEEE**, v. 3, n. 2, p. 2–15, Second 2000. ISSN 1553-877X.

HAVELUND, K. Java pathfinder, a translator from java to promela. In: **Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking**. London, UK, UK: Springer-Verlag, 1999. p. 152–. ISBN 3-540-66499-8. Disponível em: <http://dl.acm.org/citation.cfm?id=645879.672065>.

HOARE, C. A. R. Assertions in modern software engineering practice. In: **26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment, 26-29 August 2002, Oxford, England, Proceedings**. [S.l.]: IEEE Computer Society, 2002. p. 459–462.

HOLZMANN, G. J. The model checker spin. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 23, n. 5, p. 279–295, maio 1997. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/32.588521>.

IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. **IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)**, p. 1–2793, March 2012.

JHALA, R.; MAJUMDAR, R. Software model checking. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 41, n. 4, p. 21:1–21:54, out. 2009. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1592434.1592438>.

KWIATKOWSKA, M.; NORMAN, G.; PARKER, D. PRISM 4.0: Verification of probabilistic real-time systems. In: GOPALAKRISHNAN, G.; QADEER, S. (Ed.). **Proc. 23rd International Conference on Computer Aided Verification (CAV'11)**. [S.l.]: Springer, 2011. (LNCS, v. 6806), p. 585–591.

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>.

MÜFFKE, F. **A better way to design communication protocols**. Tese (Doutorado) — University of Bristol, 2004. Disponível em: <http://www.compsci.bristol.ac.uk/Publications/Papers/2000199.pdf>.

PLAT, N.; KATWIJK, J. V.; TOETENEL, H. Application and benefits of formal methods in software development. **Software Engineering Journal**, v. 7, n. 5, p. 335–346, Sep 1992. ISSN 0268-6961.

PNUELI, A. The temporal logic of programs. In: **Proceedings of the 18th Annual Symposium on Foundations of Computer Science**. Washington, DC, USA: IEEE Computer Society, 1977. (SFCS '77), p. 46–57. Disponível em: <http://dx.doi.org/10.1109/SFCS.1977.32>.

QADIR, J.; AHMED, N.; AHAD, N. Building programmable wireless networks: An architectural survey. **arXiv preprint arXiv:1310.0251**, 2013. Disponível em: <http://arxiv.org/abs/1310.0251>.

QADIR, J.; HASAN, O. Applying formal methods to networking: Theory, techniques and applications. **arXiv preprint arXiv:1311.4303**, 2013. Disponível em: <http://arxiv.org/abs/1311.4303>.

RITCHIE, D. M. **The C Programming Language**. 1988.

SIMOVNÁK, S. Verification of communication protocols based on formal methods integration. **Acta Polytechnica Hungarica**, v. 9, n. 4, p. 117–128, 2012. Disponível em: <http://uni-obuda.hu/journal/Simonak_36.pdf>.

UPPAAL. 2014. Disponível em: <http://uppaal.org/>.

VARDI, M. Y.; WOLPER, P. An automata-theoretic approach to automatic program verification (preliminary report). In: **LICS**. [S.l.]: IEEE Computer Society, 1986. p. 332–344.

WING, J. Formal methods: Past, present, and future. In: HSIANG, J.; OHORI, A. (Ed.). **Advances in Computing Science ASIAN 98**. Springer Berlin Heidelberg, 1998, (Lecture Notes in Computer Science, v. 1538). p. 224–224. ISBN 978-3-540-65388-2. Disponível em: <http://dx.doi.org/10.1007/3-540-49366-2_17>.

WOODCOCK, J. *et al.* Formal methods: Practice and experience. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 41, n. 4, p. 19:1–19:36, out. 2009. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1592434.1592436>.

XU, S.; SAADAWI, T. Does the IEEE 802.11 mac protocol work well in multihop wireless ad hoc networks? **Communications Magazine, IEEE**, v. 39, n. 6, p. 130–137, Jun 2001. ISSN 0163-6804.

YEN, D. C.; CHOU, D. C. Wireless communication: the next wave of internet technology. **Technology in Society**, v. 23, n. 2, p. 217–226, 2001. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0160791X01000100>.

ZIOUVA, E.; ANTONAKOPOULOS, T. CSMA/CA performance under high traffic conditions: throughput and delay analysis. **Computer Communications**, v. 25, n. 3, p. 313 – 321, 2002. ISSN 0140-3664. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0140366401003693>.

APÊNDICE A – PROVA DA OBTENÇÃO DO TEMPO IDEAL REAL

Suponha um conjunto de pontos de acesso A tal que $|A| = \{i \mid i \in \mathrm{N}\}$ e considerando a existência de um oráculo que distribui *slots* (δ) de tempo de uma forma ideal, *i.e.* de forma que todos os pontos de acesso em A efetuem uma transmissão com sucesso utilizando o menor tempo total possível. A função $T : i \rightarrow \mu s$ será definida por: $T(i) = (i-1)\delta + T_s$, onde T_s é o tempo necessário para uma estação transmitir com sucesso.

Assim, é fácil verificar que:
$|A| = 1 \Rightarrow T(1) = 0\delta + T_s$, ou
$|A| = 1 \Rightarrow T(i) = (i-1)\delta + T_s$

Quando A possui mais que 1 elemento, podemos definir $T(n)$ em função de $T(n-1)$ de maneira recursiva, da seguinte forma:
$T(i) = T(i-1) + (i-1)\delta + T_s$
Assim, assuma a seguinte prova por indução:

Caso base: $|A| = 2$

O tempo mínimo necessário para duas estações transmitirem será:

$$T(2) = T(1) + \delta + T_s \tag{1.1}$$

Substituindo $T(1)$ em $T(2)$:
$T(2) = T_s + \delta + T_s$
$T(2) = \delta + 2T_s$

Hipótese indutiva: $|A| > 1 \Rightarrow T(i) = T(i-1) + (i-1)\delta + T_s$

Passo indutivo: $T(i) \rightarrow T(i+1)$

$T(i) = T(i-1) + (i-1)\delta + T_s$
$T(i+1) = T(i+1-1) + (i+1-1)\delta + T_s$

$$T(i+1) = T(i) + (i)\delta + T_s \tag{1.2}$$

De (1.1) e (1.2), temos que:
$T(i) = T(i-1) + (i-1)\delta + T_s$

c.q.d.