Um Verificador de Modelos Descritos em Redes de Autômatos Estocásticos*

Claiton M. Correa, Fernando L. Dotti, Paulo Fernandes, Eli Maruani, Lucas G. Oleksinski, Afonso Sales

[1]Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Programa de Pós-Graduação em Ciência da Computação
Av. Ipiranga, 6681 Prédio 32 – 90.619-900 – Porto Alegre – RS – Brasil

{claiton.correa, eli.maruani, lucas.oleksinski}@acad.pucrs.br,
{fernando.dotti, paulo.fernandes, afonso.sales}@pucrs.br

Abstract. *During the construction of modern systems, both the satisfaction of functional properties and the ability to assert levels of performance and availability of these systems are necessary. In this sense, several modelling formalisms rely on abstractions and tools to support both the verification of functionality and perform quantitative analysis. Stochastic Automata Networks is a formalism focused on the evaluation of performance, which is attractive both because of its set of abstractions and the potential treatment of large state space. This article describes the first version of a model checker for Stochastic Automata Networks. This model checker uses Symbolic Model Checking techniques and verifies properties written in a temporal logic called Computation Tree Logic. Its use with a classic case study is reported.*

Resumo. *Durante a construção de sistemas modernos, tanto a satisfação de propriedades funcionais quanto a habilidade de afirmar níveis de desempenho e disponibilidade destes sistemas são necessárias. Neste sentido, vários formalismos de modelagem contam com abstrações e ferramentas para apoiar tanto a verificação da funcionalidade como realizar análises quantitativas. Redes de Autômatos Estocásticos são um formalismo voltado para a avaliação de desempenho, o qual é atrativo tanto devido a seu conjunto de abstrações como pelo potencial tratamento de amplo espaço de estados. Este artigo descreve a primeira versão de um verificador de modelos descritos em Redes de Autômatos Estocásticos. Este verificador faz uso de técnicas de verificação simbólica e verifica propriedades escritas na lógica temporal Computation Tree Logic. Relato do seu uso em estudos de casos clássicos são feitos.*

1. Introdução

Sistemas computacionais são utilizados em aplicações que controlam diferentes tipos de serviços, dos mais simples aos mais complexos e críticos. A crescente demanda por sistemas com estas características fez com que importantes técnicas de validação fossem desenvolvidas para verificar corretude e confiabilidade, visto que consequências nocivas podem acontecer a partir de um mau funcionamento de um *software*. Porém, garantir

*Projeto com suporte financeiro FAPERGS e CNPq.

que um sistema ou modelo esteja correto, englobando todas as características inicialmente definidas, não é uma tarefa fácil. Nos projetos de *hardware* e sistemas complexos mais tempo e esforço são gastos na verificação do que na construção dos mesmos [Clarke et al. 1999].

A abordagem de verificação de modelos conhecida como *Model Checking* é uma técnica para verificação de sistemas concorrentes de estados finitos. Apesar desta restrição, a técnica apresenta o benefício de que toda a verificação pode ser realizada automaticamente. *Model Checking* é aplicável a muitas classes importantes de sistemas e áreas do conhecimento. Dentre elas, pode-se citar o uso de verificação para controladores de *hardware*, *softwares* críticos, busca de erros em estruturas de dados, dentre outros. Ainda, possui grandes vantagens sobre abordagens tradicionais para este problema, as quais são baseadas em simulação, testes e raciocínio dedutivo [Clarke et al. 1999].

Complementarmente, formalismos e técnicas para a avaliação de desempenho de sistemas são importantes para atestar, tão logo possível durante o projeto de tais sistemas, que os mesmos atendem aos requisitos, tipicamente temporais, mas também espaciais, inseridos aos sistemas. Redes de Autômatos Estocásticos (SAN - *Stochastic Automata Networks*) são um formalismo voltado à avaliação de desempenho que permite modelagem de sistemas com grande espaço de estados [Plateau 1985]. O formalismo SAN busca prover uma forma compacta de descrever realidades complexas, modelando sistemas como um conjunto de subsistemas quase independentes que interagem ocasionalmente. Cada subsistema é composto de autômatos, onde cada autômato possui estados e transições entre eles, que podem ser realizadas através de eventos locais e/ou sincronizados. SAN permite também a representação de dependências funcionais, onde o comportamento de um autômato pode ser dependente do estado global em que a rede se encontra.

A definição de propriedades a serem verificadas em um modelo pode ser feita utilizando-se lógicas temporais, as quais descrevem a ordem em que eventos devem ocorrer no tempo sem a introdução de tempo de maneira explícita [Baier and Katoen 2008]. A introdução de lógicas temporais para *Model Checking* veio em meados dos anos 1980 por Clarke e Emerson, permitindo que este tipo de técnica de validação seja feita de forma automatizada e eficiente [Clarke et al. 1999].

A possibilidade de verificação de propriedades funcionais de modelos SAN agrega valor aos esforços de modelagem usando SAN, possibilitando um leque maior de análises a partir de um mesmo modelo. De fato, diversos formalismos para avaliação de desempenho contam com possibilidade de verificação, tal como Redes de Petri, Álgebra de Processos e Cadeias de Markov.

Este artigo descreve a primeira versão de um verificador de modelos para SAN. A ferramenta verifica se uma propriedade descrita em *Computation Tree Logic* é satisfeita dado um modelo SAN de entrada. Este artigo descreve sucintamente Redes de Autômatos Estocásticos (Seção 2); Verificação de Modelos (Seção 3); introduz a Lógica Temporal CTL (Seção 3.1); Diagramas de Decisão Multi-Valorada (Seção 4); e por fim descreve a arquitetura da ferramenta (Seção 5), bem como, conclui o trabalho (Seção 6).

2. Redes de Autômatos Estocásticos

Redes de Autômatos Estocásticos são um formalismo para modelagem de sistemas com grandes espaços de estados [Plateau 1985], e possuem equivalência de descrição com

Cadeias de Markov [Stewart 2009]. A idéia central por trás deste formalismo é prover a modularidade, permitindo que se modelem sistemas divididos em vários subsistemas de forma quase independente e ocasional interação [Benoit et al. 2003].

Um modelo SAN é descrito por um conjunto de autômatos que são compostos por estados e suas transições. Essas transições possuem uma lista de eventos associados que podem ser locais ou sincronizantes. Eventos locais alteram o estado de um único autômato, sem interferir no comportamento dos demais enquanto eventos sincronizantes estão associados a transições de dois ou mais autômatos simultaneamente, alterando o estado local de todos os envolvidos [Fernandes et al. 1998]. Como exemplo, tem-se a rede da Figura 1, problema do jantar dos filósofos, composta por três filósofos, neste caso com apenas transições locais.

Cada evento possui uma taxa de transição, que pode ser constante ou funcional, e caso um mesmo evento possa transitar para mais de um estado, uma probabilidade é associada para cada transição possível, podendo também ser constante ou funcional. Taxas e probabilidades funcionais são funções associadas ao evento, onde a função é avaliada sobre o estado global da rede e retorna um valor numérico que decidirá sobre o disparo ou não da transição [Brenner et al. 2005]. Esta é uma das vantagens na utilização de SAN em relação a outros formalismos. Pode-se consultar, por exemplo, o estado de um autômato através da operação *st* da seguinte forma: (st Autômato == Estado). Outra função comum é *nb*, que retorna a quantidade de autômatos que se encontram em um determinado estado (nb Estado).

Através de funções, pode-se construir dependências entre autômatos já que uma transição em um autômato pode estar habilitada ou não dependendo da função associada a sua taxa, a qual leva em consideração estados de outros autômatos. A sintaxe e outras funções SAN se encontram em [Benoit et al. 2007]. Na Figura 1, o evento F4 apresenta um exemplo de taxa funcional, onde a transição somente ocorrerá caso o estado do autômato Phil3 for diferente de “Right”.

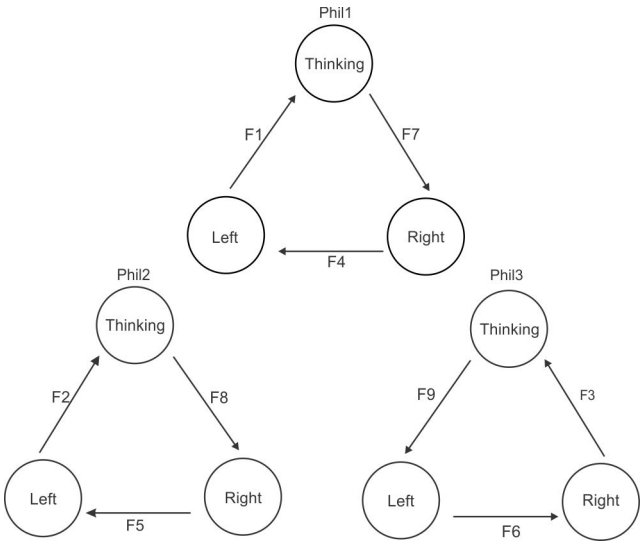
Em um modelo SAN é necessário definir uma função de atingibilidade descrevendo no modelo um ou mais estados que são conhecidamente atingíveis, pois através destes será calculado o espaço de estados atingível.

Além do uso de funções, a simplicidade e similaridade com Cadeias de Markov [Brenner et al. 2007] e o uso da álgebra tensorial generalizada também são vantagens ao se trabalhar com Redes de Autômatos Estocásticos.

Para compreensão da Figura 1, é importante conhecer o significado dos estados no modelo. Quando um filósofo quer comer, pega o garfo que está a sua direita, caso destro, ou à esquerda, caso canhoto, caracterizando a transição do estado “pensando” para o estado “quer comer”. Ao pegar o segundo garfo passa para o estado “comendo”. Nesse exemplo, modelo com três filósofos, apenas o último filósofo, filósofo 2, é canhoto.

3. Verificação de Modelos (*Model Checking*)

Model Checking pode ser definida como uma técnica automática que objetiva verificar se uma propriedade é satisfeita ou não por um modelo. Para esta abordagem, deve-se abstrair um sistema, ou seja, modela-lo em um sistema de transições de estados finitos, e descrever propriedades a serem verificadas em lógicas temporais, abordadas na Seção



TIPO	EVENTO	TAXA
LOCAL	F1	α
LOCAL	F2	α
LOCAL	F3	α
LOCAL	F4	(ST Phil3 != Right) * λ
LOCAL	F5	(ST Phil1 != Right) * λ
LOCAL	F6	(ST Phil1 != Left) * λ
LOCAL	F7	(ST Phil2 != Left) * λ
LOCAL	F8	(ST Phil3 != Left) * λ
LOCAL	F9	(ST Phil2 != Right) * λ

partial reachability = ((nb Thinking) == 3)

Figura 1. SAN que representa o problema do jantar dos filósofos para três filósofos.

3.1.

De acordo com [Clarke et al. 1999], o problema de *Model Checking* é definido como dado um modelo M de um sistema descrito em um sistema de transição de estados finitos TS e uma fórmula f, verifica-se se há estados s tal que, $s \models f$.

Essa verificação é feita de forma sistemática e como resultado retorna para o usuário uma testemunha, caso a propriedade seja satisfeita, ou um contra-exemplo caso não satisfeita. Uma testemunha indica um caminho de computação onde pode se verificar a veracidade da propriedade, enquanto um contra-exemplo apresenta uma sequência de estados que indica a não satisfação da fórmula. A possibilidade de geração de contra-exemplos e testemunhas é uma das principais vantagens em relação a outras técnicas de validação, pois seu estudo fornece um conjunto muito importante de informações para depuração do sistema sob verificação.

3.1. Lógicas Temporais

Lógica temporal é um formalismo para descrever sequências de transições entre estados em um sistema reativo. Neste tipo de lógica, o tempo não é mencionado explicitamente. Ao invés disto, uma fórmula da lógica temporal define uma relação de ordem entre estados que obedecem certas propriedades [Clarke et al. 1999]. Dentre os tipos de lógica podemos citar a *Linear Temporal Logic* (LTL), que trabalha sob uma perspectiva de tempo linear e a *Computation Tree Logic* (CTL) que trabalha sob uma perspectiva de tempo ramificado, ou seja, sob uma árvore de estados ramificada, sendo esta última abordada neste trabalho.

3.1.1. Computation Tree Logic (CTL)

Computation Tree Logic (CTL) é uma lógica temporal baseada na lógica proposicional com noção discreta de tempo. O objetivo da mesma é ser uma lógica de grande expressividade para a formulação de um importante conjunto de propriedades de sistemas. É

definida em termos de uma árvore infinita de estados onde o futuro não é determinado [Baier and Katoen 2008].

As fórmulas CTL permitem que se expresse propriedades de algumas ou todas as computações que iniciem em um determinado estado. As fórmulas são compostas de no mínimo três partes: quantificadores de caminho, que podem ser o operador universal $\forall$ (pronunciado "para todos os caminhos") ou o operador existencial $\exists$ (pronunciado "existe um caminho"), operadores temporais, que são ($\bigcirc$, **U**, $\Box$ e $\Diamond$) e/ou conectores booleanos e proposições atômicas, que são assertivas descritas sobre um modelo.

Os operadores temporais da CTL são definidos da seguinte maneira:

- $\bigcirc$ - simbolizando "no próximo estado", também representado por **X**.
- $\Diamond$ - simbolizando "em um estado futuro", também representado por **F**.
- $\Box$ - simbolizando "globalmente", também representado por **G**.
- **U** - simbolizando "até que".

Uma fórmula CTL possui sintaxe definida pela seguinte gramática:

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

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

Maiores detalhes sobre a semântica CTL são encontrados em [Hurth and Ryan 2004].

Formas normais para CTL foram criadas para se reduzir o escopo de operações a serem tratadas, o que torna menos complexa a implementação de um algoritmo de verificação, por exemplo. Através de regras de equivalência, toda fórmula CTL pode ser convertida para uma forma normal. A Forma Normal Existencial (*ENF - Existential Normal Form*) tem como característica trabalhar apenas com o quantificador de caminho existencial $\exists$, onde os operadores básicos $\exists\Box$, $\exists$U, e $\exists\bigcirc$ são suficientes para definir toda a sintaxe da CTL. Detalhes sobre a Forma Normal Existencial são encontrados em [Baier and Katoen 2008].

4. Multivalued Decision Diagrams (MDD)

Diagramas de decisão são grafos acíclicos dirigidos comumente usados para codificar de forma eficiente funções booleanas, conjuntos de estados e transições de formalismos estocásticos, ou ainda uma gama de outras funções arbitrárias [Clarke et al. 1999]. Sua aplicação está em diversos trabalhos, como verificação de circuitos aritméticos e representação de Cadeias de Markov de tempo discreto e contínuo.

Multivalued Decision Diagrams (MDD) ou Diagramas de Decisão Multi-Valorada aceitam múltiplos valores nas arestas de ligação entre os nodos e possuem as seguintes propriedades:

- Nodos são organizados em $N + 1$ níveis, onde N é o número de componentes do sistema;
- Nível N possui apenas um nodo não terminal r (nodo *root*), enquanto do nível $N - 1$ até o nível 1 há um ou mais nodos não terminais;
- Nível 0 tem dois nodos terminais: **0** e **1**;
- Um nodo não terminal p em um nível l contêm n_l arcos apontando para nodos no nível l - 1.
- Não existem nodos duplicados.

5. *Stochastic Automata Networks Model Checker*

Esta Seção descreve uma ferramenta que realiza CTL *Model Checking* para modelos descritos em SAN. Sua implementação foi feita usando linguagem de programação C++ e se baseia no algoritmo de satisfação de fórmulas CTL descrita em [Baier and Katoen 2008] (página 348). A Figura 2 apresenta um detalhamento dos processos envolvidos no verificador onde retângulos são processos e paralelogramos são dados.

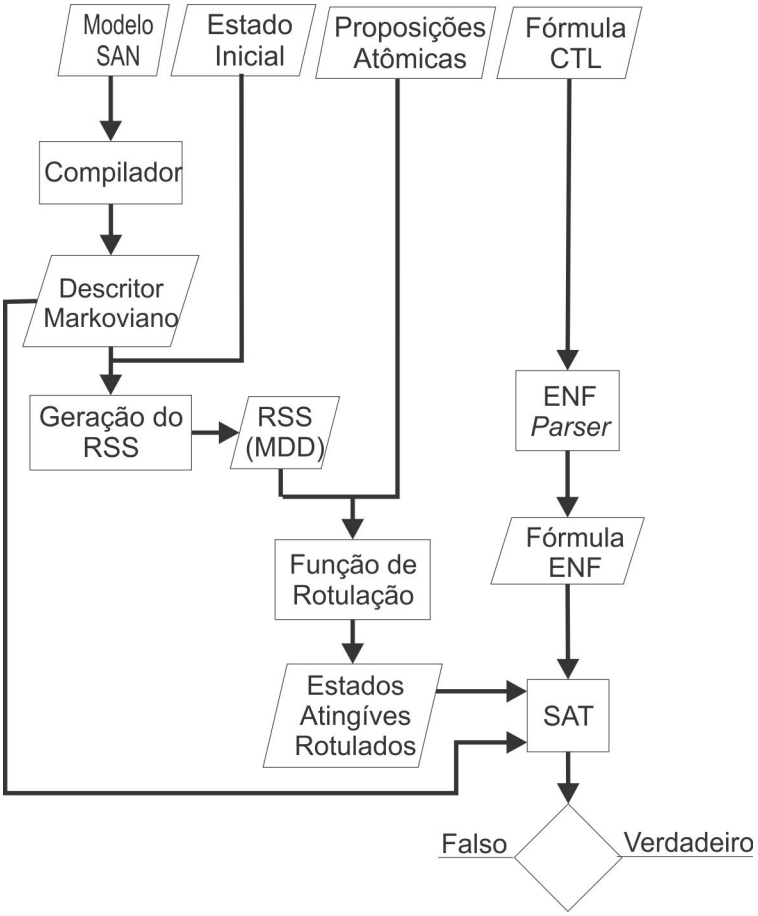


Figura 2. Arquitetura do verificador

- Modelo SAN - O modelo SAN de entrada deve ser descrito conforme abordado na Seção 2. Na ferramenta, um modelo SAN é descrito em formato textual que deve seguir as regras sintáticas descritas em [Benoit et al. 2007], tendo em vista que o compilador utilizado é o mesmo do software PEPS - *Performance Evaluation of Parallel Systems*[1].
- Estado Inicial - Um ou mais estados iniciais conhecidamente atingíveis devem ser definidos na função de atingibilidade do modelo SAN de entrada, pois esta função será o ponto de partida para a geração do RSS.
- Proposições atômicas - Proposições atômicas são expressões SAN que, assim como funções (Seção 2) podem ser avaliadas sobre o estado global da rede, retornando verdadeiro ou falso para cada estado. Elas denotam afirmações sobre a

[1] Acesso em: http://www-id.imag.fr/Logiciels/peps/

rede e são definidas no modelo SAN de entrada conforme exemplos apresentados na Seção 5.1.

- Fórmula CTL - Através de fórmulas CTL para expressões SAN são descritas propriedades sobre o modelo de entrada, conforme exemplos apresentados na Seção 5.1.
- Compilador - Após a entrada dos dados, a compilação do modelo SAN é iniciada, gerando o Descritor Markoviano que tem o objetivo de representar implicitamente as transições da Cadeia de Markov subjacente. Este descritor é que um conjunto de tensores que, ao serem operados por álgebra tensorial, permitem que se alcance todos os estados da Cadeia de Markov de maneira implícita, sendo assim extremamente eficiente para manipulação e armazenamento. As operações que podem ser efetuadas (soma tensorial ou produto tensorial) reproduzem as interações e como os diferentes autômatos sincronizam o disparo de eventos [Czekster 2010].
- Geração do RSS - O Descritor Markoviano gerado pela compilação é utilizado como Função de Transição do sistema. Logo, dado um estado qualquer do modelo, é verificado nos tensores do Descritor se há eventos locais ou sincronizados que podem ser disparados a partir deste dado estado. Se o evento tiver uma taxa constante, logo ele pode ser disparado e então o estado que é atingível a partir do dado estado é contabilizado (no caso, este estado atingível é incluído no RSS que está sendo calculado). Como modelos Markovianos estruturados possuem o conhecido problema da explosão do espaço de estados, esse processo se torna oneroso pois conforme cresce o número de autômatos e/ou estados desses autômatos, cresce exponencialmente o espaço de estados gerado, tornando problemático também seu armazenamento. Para armazenar o RSS de forma mais compacta, foi utilizado MDDs para codificar o espaço de estados, pois oferecem uma solução melhor manipulável e de armazenamento econômico. Para otimizar o processo de geração do RSS a técnica de saturação foi empregada. Essa técnica dispara exaustivamente, em um nodo de um MDD em um nível k, todos os eventos ativos que afetam somente esse nível k e abaixo, até que o conjunto de estados codificado pelo nodo $\langle k.p \rangle$, onde p é o índice do nodo no nível k, alcance um ponto fixo. A estratégia é aplicada partindo do nível mais inferior, subindo em direção ao nodo *root*. Assim, cada nodo é saturado somente uma vez após todos os nodos abaixo terem sido saturados. Como resultado desse processo temos um MDD que codifica todos os estados atingíveis do modelo SAN de entrada. Mais detalhes da geração do RSS se encontram em [Sales and Plateau 2009].
- Função de Rotulação - Esta etapa tem o objetivo de rotular todos os estados com as proposições atômicas que os satisfazem. Dados o conjunto de proposições atômicas e o RSS, a função cria de forma iterativa para cada proposição atômica do modelo, um MDD que codifica os estados do RSS onde a proposição é verdadeira.
- ENF *Parser* - Outro componente da arquitetura do verificador é o ENF *Parser*. Seu objetivo é, dada uma fórmula CTL de entrada, através de regras de equivalência [Baier and Katoen 2008], convertê-la em uma formula ENF. Sua implementação percorre em pré-ordem a árvore sintática que representa a fórmula e, não estando em Forma Normal Existencial, cria-se uma nova árvore sintática substituindo as operações fora deste escopo por outra sequência de operações equivalentes que respeita a sintaxe ENF.
- Algoritmo de satisfação (SAT) - Este algoritmo é responsável pelo processo de

verificação da fórmula de entrada sobre o modelo. Sua implementação se baseou no algoritmo de satisfação de formulas CTL em Forma Normal Existencial descrito em [Baier and Katoen 2008]. O algoritmo avalia se a fórmula de entrada é satisfeita ou não pelo modelo resultando verdadeiro ou falso. Os detalhes do algoritmo e sua implementação estão descritos na Seção 5.2.

5.1. Propriedades CTL para SAN

Definir propriedades CTL para SAN exige o esforço de expressar o que se quer saber sobre um modelo através de ambas as sintaxes. Nesse caso, descrevemos propriedades através de fórmulas CTL com proposições atômicas expressas em SAN. Proposições atômicas são asserções sobre um modelo que avaliam verdadeiro/falso para cada estado global e/ou estados locais de um ou mais autômatos. Dessa forma, tendo-se a definição de SAN em [Benoit et al. 2007] e a sintaxe da CTL apresentada na Seção 3.1.1, define-se o conjunto de propriedades CTL para Redes de Autômatos Estocásticos.

Como exemplo, podemos definir as proposições atômicas abaixo:

Phi0NaoCome = (st P0 != Left);

GarfoPhi0Phi2 = ((st P0 == Left) && (st P2 == Right));

GarfoPhi2Phi1 = ((st P2 == Left) && (st P1 == Right));

GarfoPhi1Phi0 = ((st P1 == Left) && (st P0 == Right));

NinguemCome = ((st P0 != Left) && (st P1 != Left) && (st P2 != Right));

Com as proposições atômicas acima, definidas para o modelo SAN do Jantar dos Filósofos, pode-se criar fórmulas CTL a serem verificadas, como as exemplificadas abaixo:

- $\exists\Box$(Phi0NaoCome) - Existe um caminho possível onde em todos os estados o filósofo 0 não come. Essa propriedade verifica se há a possibilidade do filósofo 0 jamais atingir o estado “comendo”. Teste de Postergação Indefinida.
- $\exists\Diamond$(GarfoPhi0Phi2 || GarfoPhi2Phi1 || GarfoPhi1Phi0) - Existe um caminho possível onde eventualmente filósofos vizinhos utilizam o mesmo garfo. As proposições atômicas usadas nesta propriedade testam todas as possíveis combinações de filósofos vizinhos utilizando o mesmo garfo. Sendo essa propriedade falsa temos um modelo mutuamente exclusivo. Teste de Exclusão Mútua.
- $\exists\Box$(NinguemCome) - Existe um caminho onde em todos os estados nenhum filósofo come. Esta propriedade verifica a possibilidade de haver caminhos onde nenhum filósofo atinge o estado “comendo”. Sendo a fórmula satisfeita o modelo possui Deadlock. Teste de Deadlock.

Vale lembrar que a primeira propriedade, Postergação Indefinida, é um exemplo utilizando apenas o filósofo 0. Caso se deseja verificar a existência de postergação indefinida em sua completude, é necessário testar para todos os filósofos do modelo.

5.2. Algoritmo de satisfação de fórmulas CTL

Esta Seção apresenta o funcionamento do algoritmo de satisfação de fórmulas CTL em ENF proposto em [Baier and Katoen 2008] e sua implementação na ferramenta apresentada neste trabalho. Como entrada, o algoritmo precisa de um sistema de transição finito

TS, um conjunto de estados S e uma fórmula CTL escrita em ENF tendo como saída o conjunto de estados que satisfazem a fórmula. A Figura 3 apresenta o algoritmo e a Figura 4 apresenta sua implementação na ferramenta.

Input: finite transition system TS with state set S and CTL formula Φ in ENF
Output: $Sat(\Phi) = \{ s \in S \mid s \models \Phi \}$
switch(Φ):

true	:	**return** S;
a	:	**return** $\{ s \in S \mid a \in L(s) \}$;
$\Phi_1 \wedge \Phi_2$	:	**return** $Sat(\Phi_1) \cap Sat(\Phi_2)$;
$\neg\Psi$	:	**return** $S \setminus Sat(\Psi)$;
$\exists\bigcirc \Psi$	:	**return** $\{ s \in S \mid Post(s) \cap Sat(\Psi) \neq \varnothing \}$;
$\exists(\Phi_1 \,\mathsf{U}\, \Phi_2)$	:	$T := Sat(\Phi_2)$; (* compute the smallest fixed point *) **while** $\{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \} \neq \varnothing$ **do** **let** $s \in \{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \}$; $T := T \cup \{ s \}$; **od**; **return** T;
$\exists\Box \Phi$	:	$T := Sat(\Phi)$; (* compute the greatest fixed point *) **while** $\{ s \in T \mid Post(s) \cap T = \varnothing \} \neq \varnothing$ **do** **let** $s \in \{ s \in T \mid Post(s) \cap T = \varnothing \}$; $T := T \setminus \{ s \}$; **od**; **return** T;

end switch

Figura 3. Função Recursiva de satisfação de fórmulas CTL [Baier and Katoen 2008].

Abaixo segue de forma detalhada a explanação do algoritmo, Figura 3, associado a sua implementação, Figura 4. O algoritmo de satisfação de fórmulas CTL é recursivo e, para cada fórmula, executa as seguintes operações:

- se a fórmula for o booleano *true* retorna o espaço de estados atingível, ou seja, S. Nesse caso, ao se avaliar no código *switch*, essa operação não será encontrada no mapeamento CTL e então executará o *default*, linha 120 da Figura 4, que verificará se o conteúdo do nodo da árvore sintática em avaliação é "T", e sendo assim retorna o MDD do RSS.
- se a fórmula for uma proposição atômica "a", retorna todos os estados do RSS rotulados por "a". Na implementação, assim como no caso anterior, executará o *default*, porém não será "T", fazendo com que o MDD receba o conjunto de estados rotulados pela proposição atômica.
- se for uma conjunção ($\wedge$) realiza a intersecção das execuções de Sat(Φ_1) e Sat(Φ_2), pois ambos precisam ser verdadeiros. Na implementação equivale a entrada na linha 14 da Figura 4.
- se for uma negação ($\neg$), o algoritmo realiza a diferença entre o RSS e a execução do Sat(Ψ), retornando todos os estados do espaço de estados atingível que não satisfazem Ψ. Na implementação equivale a entrada na linha 25 da Figura 4.

- se a fórmula for $\exists \bigcirc \Psi$ retorna o conjunto de estados onde para cada estado a intersecção de seus posteriores com Sat(Ψ) for diferente de vazio, pois caso contrário, a fórmula não é verdadeira para o estado em avaliação. Pode-se visualizar este comportamento no bloco de código da linha 32 da Figura 4, onde é executado o Sat da subfórmula a esquerda na árvore sintática e, na sequência, percorre iterativamente o RSS verificando a existência de estados sucessores para o estado em avaliação. Após, realiza a intersecção com o resultado do Sat da subfórmula executado no início e, caso essa intersecção não for conjunto vazio, o estado é adicionado ao MDD de saída.
- Uma fórmula $\exists(\Phi_1 \mathbf{U} \Phi_2)$, sendo Φ_2 verdadeiro, a mesma é satisfeita. Essa definição estabelece o menor ponto-fixo que é calculado e armazenado em um conjunto T. Partindo disto, para cada estado que satisfaça Φ_1 e ainda não esteja em T, caso a intersecção de seus posteriores com T não for vazia, inclui o mesmo em T. Ao término dessa execução, retorna o conjunto T. Este comportamento é visualizado no bloco de código da linha 50 da Figura 4, onde executa o Sat de ambas as subfórmulas de EU e armazena seus resultados. Iterativamente, realiza a diferença entre os dois resultados armazenados e verifica se o resultado é diferente de vazio. Caso verdade, realiza outra iteração percorrendo todos os estados do RSS e, para cada estado, recupera seus posteriores, realiza a intersecção com o conjunto de estados resultante de Sat da subfórmula a direita, e caso não resulte vazio, acrescenta o estado neste conjunto. A execução encerra quando não houver mais estados a serem acrescentados ao MDD de saída.
- caso a fórmula for $\exists\Box\Phi$, o seu maior ponto-fixo será o conjunto de estados que satisfazem Φ, sendo estes armazenados em T. Para cada estado em T, caso a intersecção entre seus posteriores e T for vazia, remove o mesmo de T. Ao término dessa execução, retorna o conjunto T. O bloco de código da linha 91 da Figura 4 é responsável por este comportamento. A execução inicia realizando a satisfação da subfórmula de EG seguida de um laço de execução. A cada iteração, verifica se o conjunto saída, inicialmente populado pelo resultado anterior, é diferente de vazio. Caso verdade, outra iteração percorrendo o RSS é iniciada, onde para cada estado recupera seus posteriores, realiza a intersecção com o conjunto saída e, caso resulte vazio, retira o estado em avaliação deste mesmo conjunto. A execução encerra quando não mais houver estados a serem retirados.

No código da figura 4, linha 1, tem-se como parâmetro de entrada a fórmula CTL e como parâmetros de saída os ponteiros para o diagrama de decisão com o espaço de estados que satisfaz a fórmula de entrada. $ReachSS$ é também entrada para esta função mas trata-se de uma variável global já inicializada com o espaço de estados atingível do modelo (vide Geração do RSS na Figura 2). A biblioteca de diagramas de decisão oferece várias operações sobre conjuntos de estados. Assim:

- ReachSS.Levels(): retorna o número de níveis do diagrama;
- ReachSS.RootNode(Level &kp, NodeIdx &p): retorna o nível e o índice do nodo *root* do MDD que codifica o RSS;
- ReachSS.Difference(Level kp, NodeIdx p, Level kq, NodeIdx q, Level &kr, NodeIdx &r); realiza a diferença dos MDD's codificados em <kp:p> e <kq:q> e armazena no MDD codificado em <kr:r>;
- ReachSS.Set(aut_st &tuple, int value, Level &kr, NodeIdx &r): Cria um MDD em <kr:r> com o estado em "tuple";

- ReachSS.Union(Level kp, NodeIdx p, Level kq, NodeIdx q, Level &kr, NodeIdx &r): realiza a união dos MDD's codificados em <kp:p> e <kq:q> e armazena no MDD codificado em <kr:r>;
- ReachSS.Intersection(Level kp, NodeIdx p, Level kq, NodeIdx q, Level &kr, NodeIdx &r) realiza a intersecção dos MDD's codificados em <kp:p> e <kq:q> e armazena no MDD codificado em <kr:r>;
- ReachSS.NextStates(aut_st &state, Level &kr, NodeIdx &r): retorna no MDD codificado por <kr:r> todos os estados sucessores de "state";
- ReachSS.State(Level kr, NodeIdx r, bint idx, aut_st &state): dado um MDD codificado em <kr:r> e um estado em "state", retorna o índice geral do estado em "idx";

5.3. Exemplificação

Para exemplificar o uso da ferramenta, o modelo SAN referente ao problema do Jantar dos Filósofos foi utilizado, apresentando resultados de propriedades verificadas variando o número de filósofos do modelo de entrada.

Nesta exemplificação, três clássicas situações em sistemas distribuídos foram verificadas: *deadlock*, postergação indefinida e exclusão mútua. Para tanto, fez-se necessária a definição de propriedades CTL que permitam verificar tais comportamentos. A especificação e explicação dessas propriedades é encontrada na Seção 5.1. Para a realização dos testes foi utilizado um ambiente com processador Intel Core 2 Duo 2.1 GHz com 2 MB de cache L2, 3 GB de memória RAM rodando Linux Ubuntu 10.04 32 bits. Na Tabela 1 é informado o tamanho do espaço de estados atingível para cada modelo verificado enquanto que na Tabela 2 são apresentadas as propriedades verificadas bem como o resultado, tempo de verificação e memória consumida para cada caso.

Tabela 1. Espaços de Estados Atingíveis para os modelos verificados.

Quantidade de Filósofos	RSS
3	12 estados
5	70 estados
7	408 estados
9	2378 estados
11	13860 estados
13	80782 estados
14	195025 estados

As execuções da Tabela 2 demonstram que esta primeira versão da ferramenta cumpre bem seu objetivo para pequenos modelos. Já para modelos maiores se faz necessário um ambiente mais robusto.

Percebe-se o quanto oneroso o processo de verificação é analisando a execução do modelo com 14 filósofos (RSS de 195.025 estados). Verifica-se, pelas informações da Tabela 2, que algumas propriedades puderam ser verificadas no ambiente utilizado, porém a propriedade de Exclusão Mútua o verificador precisou de mais memória do que o ambiente fornecia, ocorrendo neste caso a não verificação da propriedade. O crescimento do espaço de estados aumenta significativamente o tempo e consumo de memória de uma verificação, sendo necessária a busca por opções de otimização da ferramenta e uso de máquinas com maior poder de processamento e memória.

```
void Sat(TreeNode *formula, Level &kp, NodeIdx &p){
  char op[3];
  aut_st state(ReachSS.Levels());
  Level kq, kr, ks;
  NodeIdx q, r, s;
  bool change;

  //pega o valor que esta na raiz da arvore
  strcpy(op, formula->info);
  kp = 0; p = 0;

  //verifica o valor da raiz da arvore
  switch(CtlMap.ctlMap[op]){
    case 10://caso ^, ou seja, AND
      {
        Sat(formula->left, kq, q);

        if(q != 0){
          Sat(formula->right, kr, r);
          if(r != 0)
            ReachSS.Intersection(kq, q, kr, r, kp, p);
        }
      }
        break;
      case 7://caso ¬, ou seja, negacao
          //Guarda em <kq:q> o RSS e em <kr:r>
          ReachSS.RootNode(kq, q);
          Sat(formula->left, kr, r);

          ReachSS.Difference(kq, q, kr, r, kp, p);
          break;
      case 1://caso EX, ou seja, Existe Next
          Sat(formula->left, kq, q);

          //Percorre o RSS e, pra cada estado, busca os
          //posteriores, faz a interseccao com o Sat
          //da subformula e, caso o resultado da interseccao
          //nao for vazio, adiciona o estado no DD resultado
          for(bint i=0; i<ReachSS.RSS(); i++){
            ReachSS.State(i, state);
            ReachSS.NextStates(state, kr, r);
            ReachSS.Intersection(kr, r, kq, q, kr, r);

            if(r != 0){
              ReachSS.Set(state, 1, kr, r);
              ReachSS.Union(kp, p, kr, r, kp, p);
            }
          }
          break;
      case 8://caso EU, ou seja, Existe Until
        {
          Sat(formula->left, kq, q);
          Sat(formula->right, kp, p);

          //Enquanto houver estados para adicionar
          //continua no laco
          change = true;
          while(change){
            change = false;

            //Realiza a diferenca entre Sat de phi1 e o conjunto T
            //(codificado por <kp:p> onde T eh o conjunto que
            //inicialmente representa phi2 e incrementado com
            //novos estados cada vez que o estado avaliado
            //satisfaz uma condicao, sendo T o retorno)
            ReachSS.Difference(kq, q, kp, p, kr, r);

            //Caso a diferenca seja diferente de vazio,
            //percorre os estados de RSSe, para cada
            //estado, busca os posteriores e realiza
            //a interseccao deles com T
            if(r != 0){
              for(bint idx=0; idx<ReachSS.NumStates(kr, r); idx++){
                ReachSS.MTMDD::State(kr, r, idx, state);
                //Post(s) guarda em <ks:s>
                ReachSS.NextStates(state, ks, s);
                ReachSS.Intersection(ks, s, kp, p, ks, s);

                //Caso a interseccao dos posteriores com T seja
                //diferente de vazio inclui o estado em avaliacao em T
                if(s != 0){
                  ReachSS.Set(state, 1, ks, s);
                  ReachSS.Union(kp, p, ks, s, kp, p);
                  change = true;
                }
              }
            }
          }
        }
        break;
      case 2://caso EG, ou seja, Existe Globally
        Sat(formula->left, kp, p);

        //Enquanto houver estados para adicionar continua no laco
        change = true;
        while(change){
          change = false;

          //Caso T seja diferente de vazio, percorre os estados de RSS
          //e, para cada estado, busca os posteriores
          //e realiza a interseccao deles com T
          if(p != 0){
            for(bint idx=0; idx<ReachSS.NumStates(kp, p); idx++){
              ReachSS.MTMDD::State(kp, p, idx, state);
              //Post(s) guarda em <kq:q>
              ReachSS.NextStates(state, kq, q);
              ReachSS.Intersection(kq, q, kp, p, kq, q);

              //Caso a interseccao dos posteriores com T seja vazio
              //retira de T o estado em avaliacao
              if(q == 0){
                ReachSS.Set(state, 1, kq, q);
                ReachSS.Difference(kp, p, kq, q, kp, p);
                change = true;
              }
            }
          }
        }
        break;
      default:
        if(strcmp(op, "T") == 0){ //caso T, ou seja, true
          //Pega o root do DD que codifica o RSS
          ReachSS.RootNode(kp, p);
        }else{
          //Caso nao seja nenhum operacao,
          //entao e  uma proposição atômica
          int value = atoi(op);
          kp = kFunc[value];
          p = pFunc[value];
        }
        break;
  }
}
```

Figura 4. Código-fonte da rotina SAT.

6. Conclusão

Apresentamos neste artigo o estado da arte atual do desenvolvimento de um verificador simbólico de modelos descritos em Redes de Autômatos Estocásticos.

Tabela 2. Exemplificação de Verificação de Modelos e Propriedades.

Modelo: Jantar dos Filósofos	Tempo (s)	Memória
Postergação Indefinida (Satisfeita)		
3 Filósofos	0.035	836 KB
5 Filósofos	0.045	1.4 MB
7 Filósofos	0.075	4.8 MB
9 Filósofos	0.70	27.6 MB
11 Filósofos	4.43	170.5 MB
13 Filósofos	28.63	809.7 MB
14 Filósofos	300.38	2.2 GB
Exclusão Mútua (Insatisfeita. Modelo mutuamente exclusivo)		
3 Filósofos	0.035	832 KB
5 Filósofos	0.043	1.45 MB
7 Filósofos	0.085	5.0 MB
9 Filósofos	0.64	29.8 MB
11 Filósofos	3.38	192.9 MB
13 Filósofos	59.29	1.2 GB
14 Filósofos	-	-
Deadlock (Insatisfeita. Não há deadlock)		
3 Filósofos	0.032	844 KB
5 Filósofos	0.042	1.4 MB
7 Filósofos	0.090	4.3 MB
9 Filósofos	0.40	18.9 MB
11 Filósofos	2.83	96.9 MB
13 Filósofos	21.52	551.1 MB
14 Filósofos	43.37	675.4 MB

Em uma realidade em que sistemas computacionais estão presentes nas mais diversas aplicações, técnicas de validação de sistemas, como a verificação de modelos, são importantes para garantir propriedades fundamentais para estes sistemas, como disponibilidade, segurança de funcionamento (*safety*), entre outros. Neste contexto, a ferramenta em desenvolvimento contribui para verificação dessas propriedades em modelos de sistemas descritos em SAN, formalismo ainda não contemplado por nenhum verificador.

Através dos testes apresentados na Seção 5.3, pode-se concluir que o desempenho da ferramenta ficou limitado a modelos pequenos, principalmente no que tange a memória necessária, visto que o aumento no número de estados acarreta em um crescimento exponencial do consumo de memória pelo verificador.

7. Trabalhos Futuros

Como trabalhos futuros, objetiva-se o desenvolvimento e geração de contra-exemplos para a ferramenta em construção. Isto se faz necessário dada a importância que os mesmos apresentam para verificação de modelos e pelo poder que tem em descrever um comportamento incorreto, permitindo identificar a falha na modelagem. Para otimização da ferramenta, objetiva-se o estudo e implementação da paralelização do algoritmo de satisfação de fórmulas, o emprego da técnica de Saturação para *Model Checking* e a utilização de mecanismos de *Garbage Collection*.

Referências

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

Benoit, A., Brenner, L., Fernandes, P., Plateau, B., Sbeity, I., and Stewart, W. J. (2007). Peps 2007 user manual.

Benoit, A., Brenner, L., Fernandes, P., Plateau, B., and Stewart, W. J. (2003). The peps software tool. In *Computer Performance Evaluation/TOOLS*, pages 98–115.

Brenner, L., Fernandes, P., Plateau, B., and Sbeity, I. (2007). Peps 2007 - stochastic automata networks software tool. *Quantitative Evaluation of Systems, International Conference on*, 0:163–164.

Brenner, L., Fernandes, P., and Sales, A. (2005). The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations. *International Journal of Simulation: Systems, Science & Technology (IJSIM)*, 6(3-4):52–60.

Clarke, E. M., Grumberg, O., and Peled, A. D. (1999). *Model Checking*. MIT Press, Cambridge, Massachusetts.

Czekster, R. (2010). *Solução numérica de descritores markovianos a partir de reestruturações de termos tensoriais*. PhD thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Brasil.

Fernandes, P., Plateau, B., and Stewart, W. J. (1998). Efficient descriptor-vector multiplication in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414.

Hadzic, T., Hooker, J. N., O'Sullivan, B., and Tiedemann, P. (2008). Approximate compilation of constraints into multivalued decision diagrams. In *Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, CP '08, pages 448–462, Berlin, Heidelberg. Springer-Verlag.

Hurth, M. and Ryan, M. (2004). *Logic In Computer Science*. Cambridge University Press.

Miller, D. and Drechsler, R. (1998). Implementing a multiple-valued decision diagram package. In *28th IEEE International Symposium on Multiple-Valued Logic*, pages 52–57.

Plateau, B. (1985). On the stochastic structure of parallelism and synchronization models for distributed algorithms. *SIGMETRICS Perform. Eval. Rev.*, 13:147–154.

Sales, A. and Plateau, B. (2009). Reachable state space generation for structured models which use functional transitions. In *6th International Conference on the Quantitative Evaluation of Systems (QEST'09)*, pages 269–278, Budapest, Hungary. IEEE Computer Society.

Stewart, W. J. (2009). *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, USA.