UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ
CAMPUS LUIZ MENEGHEL - CENTRO DE CIÊNCIAS TECNOLÓGICAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

BRUNO HENRIQUE DE PAULO

UM MÉTODO PARA VERIFICAÇÃO FORMAL DE REGRAS DE PROXY

BANDEIRANTES–PR

2016

BRUNO HENRIQUE DE PAULO

UM MÉTODO PARA VERIFICAÇÃO FORMAL DE REGRAS DE PROXY

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual do Norte do Paraná para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Me. Wellington Aparecido Della Mura
Coorientador: Prof. Me. Luiz Fernando Legore do Nascimento

BANDEIRANTES–PR

2016

BRUNO HENRIQUE DE PAULO

UM MÉTODO PARA VERIFICAÇÃO FORMAL DE REGRAS DE PROXY

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual do Norte do Paraná para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Prof. Me. Wellington Aparecido Della Mura
Universidade Estadual do Norte do Paraná
Orientador

Prof. Me. Luiz Fernando Legore do Nascimento
Universidade Estadual do Norte do Paraná

Profa. Caroline Subirá Pereira
Universidade Estadual do Norte do Paraná

Bandeirantes–PR, 13 de Dezembro de 2016

"As vezes são as pessoas que ninguém espera nada
que fazem as coisas que ninguém consegue imaginar."
Alan M. Turing

AGRADECIMENTOS

A Deus pois sem ele nada existiria.

Ao meu orientador, ME. Wellington Aparecido Della Mura, pelas conversas, as quais contribuíram para para que alcançar meus objetivos neste trabalho, pelo apoio quando eu pensava que nada iria dar certo, por me ajudar a encontrar o caminho .

Ao professor Me. Carlos Eduardo Ribeiro(Biluka), pelo apoio durante os anos de graduação que contribuíram para meu desenvolvimento .

A meus pais Adriani e Marcio pelas oportunidades que me proporcionaram para meu crescimentos, pelos seus incentivos, por seus esforços que sem eles não me proporcionariam está grande oportunidade.

A minha irmã Sandy por seu apoio e força durante esses anos.

À minha namorada Tais Cristina pela apoio, amor e compreensão nos momentos que mais precisei.

Aos amigos Gabriel Petrini(Fada), Wesley Humberto(Dalsin), Fernando Moribe, Erick Kleim, Felipe Moskado, Luan Melo, Letícia Souza, Janaina Aoki, Andressa Vargas e a todos que estiveram comigo decorrer da graduação, pela ajuda e apoio durante os estudos.

A todos os professores por me proporcionarem e compartilharem seus conhecimentos no decorrer destes 4 anos.

Enfim, agradeço a todos os amigos que contribuíram direta e indiretamente para a conclusão deste trabalho.

DE PAULO, B. H.. **Um método para verificação formal de regras de proxy**. 66 p. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual do Norte do Paraná, Bandeirantes–PR, 2016.

RESUMO

A utilização de servidores *proxy* para o controle de acesso à *internet* é essencial para garantir segurança aos usuários e ao administrador do sistema. Contudo, a configuração de regras que permitam o acesso ao mesmo tempo que evitem abusos se torna um desafio em uma grande rede. A partir dessa situação, este trabalho apresenta um método de verificação das configurações de servidores *proxy* utilizando verificação formal baseada em modelos. O emprego de métodos formais para auxiliar a solução do problema torna o processo mais confiável pois as técnicas aplicadas podem ser provadas utilizando mecanismos rígidos e confiáveis. Para que a verificação das regras seja possível, inicialmente é necessário modelar as configurações de acesso do servidor *proxy*. A partir deste modelo são descritas as propriedades que serão verificadas em lógica formal e um resultado é obtido de forma que as propriedades satisfaçam o modelo gerado. Para demonstrar a efetividade da solução, foi implementada uma ferramenta para verificação automática das regras de *proxy* a partir do método proposto. Foi efetuada uma demonstração da execução da ferramenta aplicando um estudo de caso, para que possa ser observado o seu funcionamento.

Palavras-chave: Verificação de modelos, proxy, lógica temporal

DE PAULO, B. H.. **A method for formal verification proxy rules**. 66 p. Final Project (Bachelor of Science in Computer Science) – State University Northern of Parana , Bandeirantes–PR, 2016.

ABSTRACT

The use of proxy servers for internet access control is essential to ensure the security of users and system administrator. However, setting up rules that allow access while avoiding abuse becomes a challenge on large networks. From this situation, this work presents a method of checking the configurations of proxy servers using formal verification. The use of formal methods to help solve the problem makes the process more reliable because the techniques used can be proved using rigid and reliable mechanisms. To reach the rule's verification, we must first model the proxy settings for the proxy server. From this model are described the properties that will be verified in formal logic and a result is obtained so that the properties satisfy the model generated. To demonstrate the effectiveness of the solution, a tool was implemented to automatically verify the rules of a proxy using the proposed method. A demonstration of the use of the tool was made using a case study, so that its operation can be observed.

Keywords: Model checking, proxy, temporal logic.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de funcionado de um *proxy* [1] 23
Figura 2 – Modelo de Transições possíveis do programa [2] 33
Figura 3 – Exemplo de código em Uppaal [3] 36
Figura 4 – Modelo Grafico do Train em Uppaal [3] 36
Figura 5 – Cadeia Complexa [5] 39
Figura 6 – Semântica das Propriedades. 46
Figura 7 – Exemplo de configuração. 48
Figura 8 – Modelo 49
Figura 9 – Funcionamento da ferramenta 51
Figura 10 – Diagrama de pacotes da ferramenta 52
Figura 11 – Diagrama de sequencia do analisador sintático 53
Figura 12 – Classes de representação e construção do autômato 53
Figura 13 – Classes de representação da simulação do autômato 54
Figura 14 – Código-fonte do construtor *ConstruirAutomato*. 56
Figura 15 – Exemplo Configuração Squid. 57
Figura 16 – Exemplo de Autômato construído *ConstruirAutomato*. 58
Figura 17 – Exemplo de entrada para verificação. 58
Figura 18 – Exemplo de Saida. 58
Figura 19 – Arquivo de Configuração da Empresa. 60
Figura 20 – Casos de Teste. 61

LISTA DE ABREVIATURAS E SIGLAS

ACL	*Access Control List* (Lista de Controle de Acesso)
BDD	*Binary Decision Diagram* (Diagrama de decisão Binária)
CTL	*Computer Tree Logic* (Lógica de Computação em Árvore)
LTL	*Linear Temporal Logic* (Lógica Temporal Linear)
FTP	*File Transfer Protocol*(Protocolo de Transferência de Arquivos)
HTTPS	*Hyper Text Transfer Protocol Secure*(Protocolo de Transferência de Hipertexto Seguro)

SUMÁRIO

1 INTRODUÇÃO 19
1.1 Justificativa 19
1.2 Objetivos 20
1.2.1 Objetivo Geral 20
1.2.2 Objetivos Específicos 20
1.3 Organização do Trabalho 20

2 FUNDAMENTAÇÃO TEÓRICA 23
2.1 Proxy 23
2.1.1 Squid 24
2.2 Model Checking 26
2.3 Lógicas 26
2.3.1 Lógica Proposicional 27
2.3.2 Lógica Modal 28
2.3.3 Lógica Temporal 29
2.3.3.1 Lógica Temporal Linear 29
2.3.3.2 Lógica de Computação em Árvore 31
2.4 Ferramentas para Model Checking 32
2.4.1 NuSMV 32
2.4.2 Spin 33
2.4.3 Uppaal 35

3 TRABALHOS RELACIONADOS 37
3.1 Verificação Formal aplicado a políticas de configuração de Firewall 37
3.2 Fireman uma Ferramenta para Análise e Modelagem de Firewall 38

4 ESTRUTURAÇÃO DA PESQUISA 41

5 MÉTODO PROPOSTO 43
5.1 Modelo do Proxy 43
5.2 Definição das Características do *Proxy* 45
5.3 Aplicação do Método 48

6 FERRAMENTA DE VERIFICAÇÃO 51
6.1 Análise e Projeto 52
6.1.1 Construção do Autômato 52

6.1.2 Simulação do Autômato 54
6.2 Desenvolvimento 54

7 ESTUDO DE CASO 57
7.1 Aplicação da Ferramenta 57
7.1.1 Descrição do Caso 58
7.1.2 Aplicando a Ferramenta no Caso 59

8 CONCLUSÃO 63

REFERÊNCIAS 65

1 INTRODUÇÃO

A internet se tornou o maior ambiente global de distribuição, troca e compartilhamento de informações e com isso ficou comum em variados lugares, como ambiente de trabalho, estudo e lazer. Muitas vezes este uso precisa ser limitado para que certos recursos da web não possam ser acessados em determinada hora e local, como em locais de trabalho ou estudo para que os usuários não percam o rendimento. Para possibilitar esta limitação geralmente são empregados servidores *proxy*, que atuam entre o usuário e a sua requisição na internet[6].

Servidores *proxy* são aplicados para controlar otimizar a rede e armazenar endereços acessados por um domínio de usuários[7]. Um *proxy* também pode auxiliar provedores de internet para cumprir normas exigidas pela legislação brasileira, como a obrigação de guardar registros de acesso à internet por um período mínimo determinado pela lei de modo que possam ser utilizados pela justiça se necessário[8]. Apesar das várias funcionalidades de uso do servidor *proxy* o enfoque deste trabalho está relacionado com a verificação de suas regras de funcionamento, que é a parte de sua configuração que limita o acesso a determinados acessos na *web*, porém as configurações feitas por um profissional com experiência, podem apresentar erros de configuração, levando ao mau funcionamento ou a uma situação indesejada.

Qualquer tipo de sistema mesmo bem elaborado está sujeito a falhas. Um exemplo de sistema que falhou, foi o do foguete Ariane 5 que explodiu 4 segundos após ser lançado. Neste caso, os engenheiros apontaram que a falha ocorreu devido um erro de software que não foi testado adequadamente. Para evitar situações como essa, existem vários métodos para verificação de sistemas como dedução, simulação e também *Model Checking*[9]. O teste por *Model Cheking* é uma verificação onde se possui um modelo que pode descrever todas as ações do sistema e o qual precisa ser satisfeito por uma certa propriedade geralmente expressa por uma lógica formal, que é verificada se é satisfeita ou não perante o modelo descrito[9].

O objetivo deste trabalho é explicar detalhadamente como são os processos de *Model Checking* e mostrar uma maneira de aplicá-los para a verificação das configurações de *proxy*, criando um modelo e procurando a melhor maneira de expressar as configurações em lógica e qual a melhor lógica a ser utilizada.

1.1 Justificativa

Mesmo com um bom treinamento e muita experiência do administrador do *proxy*, as suas configurações ainda estão sujeitas a erros, e encontrá-los manualmente pode ser

uma tarefa exaustiva. Isso se deve ao fato de que os arquivos de configuração de *proxy* são geralmente muito extensos, complexos e cheios de armadilhas que induzem ao erro[1].

Corrigir estes erros de maneira automática e de forma ágil tornaria a vida do administrador de *proxy* mais fácil. De forma que após definir as configurações do *proxy* suas regras possam ser testadas antes de entrarem em execução.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é propor um método para a verificação automática das configurações do Squid *proxy* utilizando métodos formais para criar um modelo capaz de detectar falhas na configuração de um servidor *proxy*.

1.2.2 Objetivos Específicos

Os objetivos específicos são:

- **Estudar modelos formais que possam expressar as configurações de um *proxy*:** cada tipo de sistema pode possuir um determinado modelo distinto, para o desenvolvimento do modelo;
- **Criar um tradutor das configurações do *proxy* para o modelo formal escolhido:** criar um algoritmo que mostre como as configurações do *proxy* podem ser construídas no modelo definido;
- **Definir um formato para descrever as regras do *proxy* a serem satisfeitas pelo modelo:** encontrar um formato para expressar as propriedades do proxy desejáveis no modelo;
- **Determinar propriedades desejáveis em um *proxy* e expressá-las em logica formal:** em verificações por modelos, são necessárias propriedades a serem verificadas. Elas descreverão as características do sistema desejado e também necessárias para verificação e validação do modelo;
- **Criar uma ferramenta que ajude na verificação:** criar uma ferramenta capaz de efetuar a criação do modelo e verificar automaticamente propriedades desejáveis no modelo.

1.3 Organização do Trabalho

A organização do trabalho obedece a estrutura a seguir. No Capitulo 2 é apresentado um referencial teórico sobre *proxy*, *Model Checking*, lógicas e ferramenta para

Model Checking. No Capítulo 3 são apresentados os trabalhos mais intimamente relacionados com a pesquisa apresentada neste documento. No Capítulo 4 resume os passos a serem seguintes para que se alcance os objetivos deste trabalho. No Capítulo 5 contém o método desenvolvido neste trabalho. No Capítulo 6 descreve o planejamento para o desenvolvimento da ferramenta, assim como o desenvolvimento da ferramenta. No Capítulo 7 apresenta um estudo de caso aplicado ao uso na ferramenta desenvolvida. E finalmente, o Capítulo 8 descreve uma conclusão do método proposto, os problemas pendentes e possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta informações relacionadas com *proxy*, como seu funcionamento, suas características e modo de configuração. Também são descritos conceitos sobre *Model Checking* seu funcionamento e algumas possíveis lógicas para a descrição de propriedades de *Model Checking*.

2.1 Proxy

Um servidor *proxy* é um serviço que está entre o cliente e a página web solicitada. Como pode ser visto na Figura 1, quando o cliente solicita uma página para o servidor *proxy*, este recupera os dados do solicitante e faz a requisição da página como se fosse o cliente. Após o carregamento pelo servidor, a página solicitada é enviada para o cliente. Em um nível mais baixo o servidor *proxy* faz uma filtragem das requisições solicitadas baseadas em suas regras de acesso onde os pedidos podem ser validados ou não perante as regras preestabelecidas, as regras são geralmente baseadas no endereço de IP do cliente, servidor de destino, protocolo, tipo de conteúdo [10].

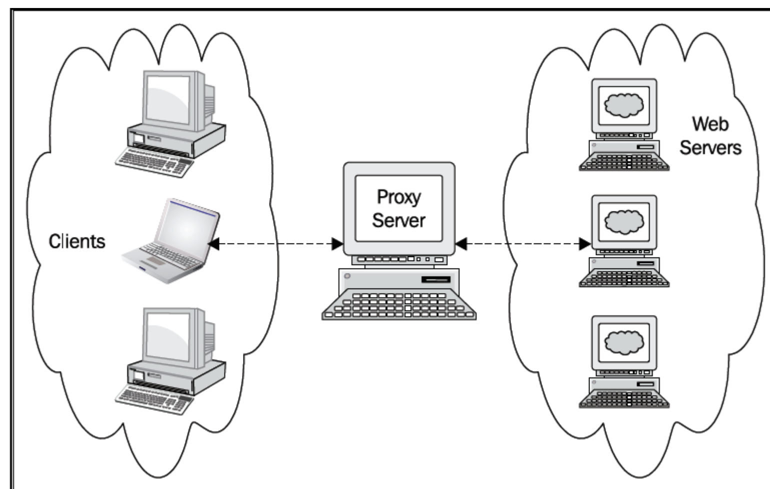

Figura 1 – Exemplo de funcionado de um *proxy* [1]

Os servidores *proxy* são geralmente utilizados para:

- Reduzir a largura de banda da rede;
- Melhorar o tempo de acesso do usuário aplicando os conceitos de cache;
- Monitorar o tráfego da rede.;
- Aplicar regras de acesso, para restringir solicitações;

- Retransmitir o tráfego em uma rede local;
- Filtrar solicitações ou respostas com sistema de detecção integrado contra vírus e *malware*.

Segundo Wessels[1], Web cache é o ato de armazenar certos recursos da web para uma possível reutilização no futuro. Semelhante ao cache de computadores, o *cache web* possui as taxas de acerto(*hit*) e erro(*miss*). O *hit* ocorre quando o servidor *proxy* consegue satisfazer a solicitação com seu cache, Essa taxa de hit em servidores *proxy* podem variar de 30% a 60%. Os erros(miss) ocorrem quando o servidor não consegue satisfazer a solicitação com o seu *cache*, como por exemplo, quando uma requisição é feita pela primeira vez.

2.1.1 Squid

O Squid é um servidor *proxy* que possue suporte em HTTPS, FTP entre outros, o Squid também possui um sistema de cache que tem como objetivo diminuir o fluxo da banda e aumentar o tempo de resposta as suas requisições. Para seu funcionamento não é necessário um *hardware* muito sofisticado, porém por se tratar de um sistema de cache o Squid se faz necessário um pouco de memória, o espaço em disco também é importante pois quanto mais espaço em disco mais espaço em cache haverá para suas requisições. O Squid faz o uso de 32MB de memória para cada GB de espaço no disco[1].

Para configurar as regras do Squid é necessário abrir o seu arquivo de configuração em um editor de texto, o arquivo de configuração geralmente se encontra na pasta de instalação do Squid `/opt/squid/etc/squid.conf` neste arquivo é onde será definido em qual porta o servidor funcionará, o tamanho do seu cache e da suas regras de controle, porém a base do funcionamento do *proxy* está totalmente relacionada com suas *Access Control Lists*(ACLs). As ACLs são declaradas da seguinte maneira: `acl [nome] [tipo] [valores]`. A seguir, são apresentados alguns exemplos de ACL's[10] :

```
acl meu_Computador_pessoal src 192.168.2.21
acl PC_secretaria src 192.168.0.25
acl destino dst 200.175.44.1
```

O nome da regra é um identificador para permitir ou proibir sua execução. O tipo da regra representa se os valores se referem ao destino (`dst`), à origem (`src`) ou à porta (`port`) da requisição, o Squid também apresenta outros tipos de regras além dos

já citados. Os valores declarados na regra podem conter o ip de um *host*, o endereço de um site,um domínio de rede e seu intervalo, dependendo do tipo da ACL[10]. Em alguns casos, os valores podem indicar um caminho que leva a um arquivo que contém todos os itens da regra quando se tratam de muitos dados à serem declarados.

Isto é, após a declaração das ACL's, elas serão combinadas com o comando http_access que é acompanhado de *deny*(negado) e *allow*(permitido) estes que serão responsáveis pelas decisões baseadas nas regras[10]:

- http_access [deny ou allow] [regra]: este comando define se a regra terá acesso ou não as transações http.

Com as regras definidas o Squid as interpreta de cima para baixo até que seja definida uma ação permitir ou negar. Se possuir muitos valores dentro de uma ACL o Squid as interpretará da esquerda para a direita até que uma delas satisfaça as suas condições [10]. Como descrito, o Squid apresenta funções abrangentes, porém o enfoque deste trabalho é apenas a verificação de suas regras e para este trabalho serão consideradas dois tipos de ACL's `dst` e `src`:

- **SRC**: As ACL do tipo `src` podem ser declaradas de duas maneiras: um endereço de ip de um cliente ou então um range de ips. Um cliente pode ser declarado da seguinte maneira (`acl cliente src 192.0.2.25/32`). Está ACL responderá à todas solicitações do cliente `192.0.2.25` a mascará do cliente precisa ser especifica quando declarado uma ACL de um único ip, porém se ela não for especificada durante a declaração da ACL o Squid tentará determiná-la automaticamente um exemplo é a ACL (`acl cliente src 192.0.2.25`) que também antederá à todos os pedidos para o cliente do ip `192.0.2.25`. Uma ACL com um range de ips pode ser declarada da seguinte maneira (`acl rede src 192.0.2.0/25`) está ACL declara um grupo de endereços de ip entre o ip `192.0.2.1` até o ip `192.0.2.127` pois denota que dos 32 bits disponíveis para atribuição de ip sobram apenas 7 bits para representar os endereços de rede que seria de `0-127` [10].

- **DST**:As ACL do tipo dst são declaradas da seguinte maneira

  `acl servidor 200.175.44.1/255.255.255.255` está regra trata o destino da navegação em uma requisição [10].

2.2 Model Checking

O termo *Model Checking* se refere a uma técnica automática para verificar os estados finitos de sistemas. Esta técnica oferece inúmeras vantagens sobre os métodos tradicionais que geralmente baseiam-se em simulação. Teste e dedução são utilizados com sucesso na verificação de circuitos complexos e em protocolos de comunicação.

Mesmo poderosas, as técnicas de *Model Checking* possuem um grande desafio quando o modelo a ser verificado é muito complexo. Por exemplo, na comunicação entre sistemas em que existe uma grande quantidade de componentes de outros sistemas ou complexas estruturas de dados, existe a possibilidade de um grande aumento do número de estados, o que gera uma explosão combinatória[9].

Os processos de *Model Checking* são divididos em três partes conforme segue:

Modelagem: busca criar um modelo do sistema que será checado e modelado em um método formal, que pode ser um automato ou modelo de Kripke ou outro da escolha de quem está modelando[9].

Especificação das propriedades: acontece antes da verificação e nesta fase são definidas as propriedades que serão verificadas pelo modelo. Geralmente as propriedades são expressas por alguma linguagem formal, como as lógicas. Nos sistemas de *hardware* e *software* são geralmente utilizada a lógica temporal, que expressa o conceito de evolução dos sistemas. Existem diversos tipos de lógicas que podem ser empregadas para as definições destas especificações como CTL*, CTL, LTL [9].

Verificação: esta fase coloca em prática tudo o que foi feito nas fases anteriores. O ideal é que a verificação seja automática, mais geralmente sempre envolve assistência humana, a verificação possui dois tipos de saída, aceito ou rejeitado. Quando é retornada uma saída aceita, isto significa que as propriedades especificadas para a verificação foram satisfeitas pelo modelo, sendo assim estão de acordo com o modelo gerado, caso as propriedades não sejam aceitas, alguns erros podem ocorrer como: as propriedades foram expressadas de maneira erronia; o modelo realmente não esta de acordo com as propriedades; ou ocorreu um erro na modelagem do sistema a ser verificado[9].

2.3 Lógicas

Segundo Huth e Ryan(2004), o uso da lógica em Ciência da Computação tem como objetivo modelar situações do mundo real encontradas pelos profissionais da área, de forma que possam raciocinar sobre elas. Formalmente e encontram maneiras que possam ser executadas de forma automática em uma máquina.

2.3.1 Lógica Proposicional

A lógica proposicional é baseada em proposições ou sentenças declarativas que podem ser verdadeiras ou falsas. Sua estrutura precisa ser expressada de forma que as frases levem a alguma estrutura lógica[2].

Segue alguns exemplos de frases:

- A soma dos números 3 e 5 é igual a 8.
- Jane reagiu violentamente a acusação de Jack.

Estas frases são todas declarativa pois todas podem assumir os valores de verdadeiro ou falso. A primeira sentença pode ser testada utilizando os conceitos de matemática básica. Já a segunda sentença seria um pouco mais complicada, pois seria necessário de um testemunho de alguém que presenciou o acontecimento[2].

Há outro exemplo onde podemos atribuir determinados símbolos como X, Y, Z a determinadas proposições, assim podemos expressar as sentenças de maneiras mais complexas[2].

- X: Eu ganhei na loteria na semana passada.
- Y: Eu comprei um bilhete de loteria.
- Z: Eu ganhei no sorteio da semana passada.

A lógica proposicional também possui alguns operadores lógicos conforme segue:

- $\neg$: A negação de X poderia ser denotada por $\neg X$ e expressaria "*Eu não ganhei na loteria semana passada*" ou é equivalente que "*Não é verdade que ganhei na loteria na semana passada*".
- $\vee$: Dado X e Z queremos afirmar que um deles é verdadeiro: "*Eu ganhei na loteria na semana passada*" ou "*Eu ganhei no sorteio da semana passada*", que também pode ser declarado por $X \vee Z$.
- $\wedge$: A proposição $X \wedge Z$ denota que para que a proposição seja verdadeira é necessário uma conjunção de verdades, por exemplo: ' '*Eu ganhei na loteria na semana passada*" e "*eu ganhei no sorteio da semana passada*".
- $\rightarrow$:Implicação pode ser expressa da seguinte maneira: "*Se Eu ganhei na loteria semana passada*" então "*Eu comprei um bilhete de loteria*", que pode ser reescrito como $X \rightarrow Y$.

- $\iff$: Implica se $X \iff Y$ só será verdadeira apenas se A e B forem falsos ou verdadeiros.

Após conhecer os operadores lógicos pode-se utilizá-los de varias formas para a construção de diferentes regras.

Como:

$(X \wedge Y) \rightarrow (\neg Z \vee Y)$[2].

O que significa que se X e Y então não Z ou Y. Estes exemplos mostram que a lógica proposicional pode ser utilizada para expressar as proposições de varias maneiras formalmente, facilitando a análise feita por máquinas.

2.3.2 Lógica Modal

A lógica modal é uma variação da lógica proposicional na qual considera a existência de mundos, ou seja, a modificação das proposições de acordo com evolução de um modelo. Na lógica modal, uma proposição é necessária se é verdadeira em todos os mundos possíveis, e é possível se é verdadeira em pelo menos um mundo[11].

A lógica modal possui a seguinte sintaxe, descrita na Forma de Backus Naur:

$\phi ::= p \mid \top \mid \bot \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \mid (\phi \rightarrow \phi) \mid (\phi \Leftrightarrow \phi) \mid \Box\phi \mid \Diamond\phi.$

Neste caso p equivale a preposições atômicas, $\top$ e $\bot$ a verdadeiro ou falso, $\Diamond$ e $\Box$ são os operadores modais de possibilidade e necessidade, respectivamente, e os demais operadores equivalem aos da lógica proposicional[12].

A semântica da lógica modal pode ser descrita utilizando um modelo de Kripke $M = (S, \rightarrow, L)$, onde S são os estados ou descritos como mundos na lógica modal. $\rightarrow$ é a relação de acesso entre os mundos, e L é a rotulação das proposições válidas nos mundos[12].

Supondo que w seja um mundo em S no modelo $M = (S, \rightarrow, L)$ a relação de satisfação pode ser definida da seguinte maneira:

1. $M, w \models p \Leftrightarrow w \in L(p)$
2. $M, w \models \top$ sempre
3. $M, w \models \bot$ nunca
4. $M, w \models \neg\phi \Leftrightarrow M, w \nvDash phi$
5. $M, w \models \phi \wedge \psi \Leftrightarrow M, w \models \phi$ e $M, w \models \psi$
6. $M, w \models \phi \vee \psi \Leftrightarrow M, w \models \phi$ ou $M, w \models \psi$

7. $M, w \models \phi \rightarrow \psi \Leftrightarrow$ Se $M, w \models \phi$ então $M, w \models \psi$

8. $M, w \models \phi \leftrightarrow \psi \Leftrightarrow M, w \models \phi$ se e apenas se $M, w \vDash \psi$

9. $M, w \models \Box\phi \Leftrightarrow \forall v \in S$ tal que $(w, v) \in \rightarrow$ então $\phi \in L(v)$

10. $M, w \models \Diamond\phi \Leftrightarrow \exists v \in S$ tal que $(w, v) \in \rightarrow$ então $\phi \in L(v)$

O item (1) reflete que a preposição p está marcada como verdadeira em um mundo w, os itens (2) e (3) são constantes sempre verdadeira ou sempre falsa no mundo S, dos itens (4) a (8) os valores são equivalentes a lógica proposicional, no item 9 o operador modal $\Box\phi$ só é satisfeito pelo modelo M se ϕ for verdadeira em todos os mundos alcançáveis por w,$\Diamond\phi$ é satisfeito por M desde que exista pelo menos um mundo alcançável por ϕ[11].

2.3.3 Lógica Temporal

A lógica temporal é muito importante para as técnicas de *Model Checking*, pois os seus modelos se baseiam nela. Um modelo de lógica temporal possui vários estados que podem assumir valores de verdadeiro ou falso. Estes valores podem sofrer modificações conforme o sistema vai evoluindo. A criação dos estados e seus valores são baseados em um modelo de Kripke denotado pela tupla $M = (S, \rightarrow, L)$, onde S é o conjunto de estados, $\rightarrow$ é a relação de transição dos estados e L é a função de rotulação dos estados quando sua proposição atômica é verdadeira[2].

Para a proposição atômica ser verdadeira ou falsa tudo dependerá das relações de transição entre os estados do modelo definido, o conjunto de estados evoluem ao longo do tempo no modelo, as proposições são avaliadas a cada estado, sendo assim a noção de verdade não é estática e sim dinâmica que evolui ao longo do tempo [2].

A lógica temporal pode se dividir em dois tipos: linear e de computação em árvore. Na linear é considerado que cada estado possui apenas um futuro possível, na ramificada cada estado pode possuir vários futuros possíveis, estas lógicas serão descritas nas próximas seções.

2.3.3.1 Lógica Temporal Linear

Uma lógica temporal linear possui uma sequencia de estados, os quais também podem ser chamados de caminhos de computação. Nestas sequencias definem os estados futuros conforme a evolução do sistema. As proposições ou fórmulas da LTL são interpretadas sobre esta sequencia de estados, as proposições podem ser da forma (p1,p2,p3,. . .), e também podem ser o estado de um sistema como ‘O Computador está ligado’, ‘O Computador está em espera. Estas escolhas de valores da preposições vão variar de acordo com o sistema que será escolhido[2].

A lógica temporal linear possui a sintaxe, descrita na Forma de Backus Naur, conforme apresenta a seguir:

$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (X\phi) \mid (F\phi) \mid (G\phi) \mid (\phi\ U\ \phi')$

Onde p é qualquer preposição atômica,$\top$ é verdadeiro e $\bot$ é falso, os operadores$\neg, \wedge, \vee \rightarrow$ são equivalentes aos da lógica proposicional e X,F,G e U são chamados conectivos temporais, $X\phi$ significa que ϕ será verdadeiro no próximo estado, $F\phi$ que ϕ será verdadeiro em algum estado no futuro do caminho, $G\phi$ que ϕ será verdadeiro para todos os estados do caminho,$\phi\ U\ \phi'$ que ϕ será verdadeiro até que ϕ' seja verdadeira no caminho[2].

Dado um modelo $M = (S, \rightarrow, L)$ e um caminho π e uma fórmula LTL b de forma que $M, \pi \models b$, onde $\models$ é a relação de satisfação ou seja, verifica-se o caminho π do modelo M satisfaz a fórmula b. Os caminhos são formados por sequências de estados S onde suas posições variam de π^1 até π^n onde n é o número de posições de π. Dado o caminho π de um modelo M as relações de satisfação podem ser descritas das seguintes maneiras:

1. $\pi \models \top$
2. $\pi \not\models \bot$
3. $\pi \models p \Leftrightarrow p \in L(s1)$
4. $\pi \models \neg\phi \Leftrightarrow \pi \not\models \phi$
5. $\pi \models \phi1 \wedge \phi2 \Leftrightarrow \pi \models \phi1$ e $\pi \models \phi2$
6. $\pi \models \phi1 \vee \phi2 \Leftrightarrow \pi \models \phi1$ ou $\pi \models \phi2$
7. $\pi \models \phi1 \rightarrow \phi2 \Leftrightarrow \pi \models \phi2$ sempre que $\pi \vDash \phi1$
8. $\pi \models X\phi \Leftrightarrow \pi^2 \models \phi$
9. $\pi \models G\phi \Leftrightarrow \forall i \geq 1, \pi^1 \models \phi$
10. $\pi \models F\phi \Leftrightarrow \exists i \geq 1, \pi^1 \models \phi$
11. $\pi \models \phi\ U\ \psi \Leftrightarrow \exists i \geq 1, \pi^1 \models \psi$ e $\forall j, 1 \leq j < i \mid \pi^j \models \phi$

As definições (1) e (2) indicam proposições verdadeiras e falsas, a definição 3 indica que p é valido no primeiro estado do caminho, as definições de (4) a (7) estão definidos os operadores da lógica proposicional, a definição (8) considera-se o próximo caminho para verificar a primeira, (9) e (10) apresentam os operadores G que apresenta que ϕ é verdadeiro em todo o caminho e F que ϕ será verdadeiro em alguma parte do caminho,finalmente a definição (11) que utiliza o operador $U\ \psi$ será verdadeiro se todas as posições anteriores ϕ for verdadeiro[2].

2.3.3.2 Lógica de Computação em Árvore

A lógica LTL é chamada de linear pois é baseada em apenas um caminho, ou seja existe apenas um estado sucessor possível, o que torna a lógica LTL limitada a uma única possibilidade possível de caminho[13]. A lógica de computação em árvore ou CTL, trata-se de uma lógica na qual apresenta todos os caminhos possíveis, não ficando limitada a apenas uma sequência única de de estados e sim trabalhando e criando uma arvore de ramificação contendo todos os caminhos[13].

A sintaxe da CTL pode ser descrita da seguinte forma em Backus Naur:

ϕ ::= $\top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (AX\phi) \mid (EX\phi) \mid (AF\phi) \mid (EF\phi) \mid (AG\phi) \mid (EG\phi) \mid A(\phi\ U\ \phi') \mid E(\phi\ U\ \phi')$

Os conectivos possuem os mesmos significados da lógica LTL, porém na lógica CTL são adicionados dois conectivos que são utilizado sempre aos pares com os outro conectivos, o conectivo A (inevitavelmente) e o conectivo E (provavelmente)[2].

A semântica da lógica CTL pode ser descrita da seguinte forma utilizando um modelo de kripke $M = (S, \rightarrow, L)$, onde S é o conjunto de estados,$\rightarrow$ as relações de transições, L são as marcações, $s \in S$ e ϕ é uma fórmula CTL.

1. $M, s \models \top$
2. $M, s \not\models \bot$
3. $M, s \models p \iff p \in L(s)$
4. $M, s \models \neg\phi \Leftrightarrow M, s \not\models \phi$
5. $M, s \models \phi 1 \wedge \phi 2 \Leftrightarrow M, s \models \phi 1$ e $M, s \models \phi 2$
6. $M, s \models \phi 1 \vee \phi 2 \Leftrightarrow M, s \models \phi 1$ ou $M, s \models \phi 2$
7. $M, s \models \phi 1 \vee \phi 2 \Leftrightarrow M, s \not\models \phi 1$ ou $M, s \models \phi 2$
8. $M, s \models AX\phi \Leftrightarrow \forall s1 \mid s \rightarrow s1$ com $M, s1 \models \phi$
9. $M, s \models EX\phi \Leftrightarrow \exists s1 \mid s \rightarrow s1$ com $M, s1 \models \phi$
10. $M, s \models AG\phi \Leftrightarrow \forall\pi = s1 \rightarrow s1 \rightarrow s2 \rightarrow s3 \ldots \mid s1 = s\ e\ \forall Si \in pi$ com $M, si \models \phi$
11. $M, s \models EG\phi \Leftrightarrow \exists\pi = s1 \rightarrow s1 \rightarrow s2 \rightarrow s3 \ldots \mid s1 = s\ e\ \forall Si \in pi$ com $M, si \models \phi$
12. $M, s \models AF\phi \Leftrightarrow \forall\pi = s1 \rightarrow s1 \rightarrow s2 \rightarrow s3 \ldots \mid s1 = s\ e\ \exists Si \in pi \mid M, si \models \phi$
13. $M, s \models AF\phi \Leftrightarrow \exists\pi = s1 \rightarrow s1 \rightarrow s2 \rightarrow s3 \ldots \mid s1 = s\ e\ \exists Si \in pi \mid M, si \models \phi$

Os itens (1) e (2) respectivamente representam se o caminho s é verdadeiro ou falso, do (3) ao (8) são utilizados os operadores ja descritos da lógica proposicional , do (9) ao (13) referem-se a caminhos de computação em modelos que são muito úteis para visualizar todos os caminhos possíveis a partir de um determinado estado[2].

2.4 Ferramentas para Model Checking

Model Checking não é uma técnica muito nova em verificação de sistemas, pesquisas em *Model Checking* são desenvolvidas desde os anos 80, durante esses anos foram desenvolvidas ferramentas para descrever modelos e verificar sistemas os quais cada um dele possui seu método de entrada lógica para a verificação[3].

Nas próximas seções serão descritas algumas ferramentas para *Model Checking* e como elas funcionam.

2.4.1 NuSMV

A verificação por modelos pode ser um assunto prático, pois existem varias ferramentas que podem ser utilizadas para a verificação de sistemas, uma dessas ferramentas é o NuSMV(*New Symbolic Model Verifier*) ou Novo Modelo Simbólico de verificação, é uma ferramenta Open Source(Código Aberto)e possui um bom número de usuários. Esta ferramenta fornece uma linguagem para descrever um modelo em diagrama e verificar a validade de suas fórmulas LTL ou CTL[2].

Programas NuSMV consistem de funções como nas linguagens C e Java, uma delas precisa ser a principal(*main*), podendo ser declaradas variáveis e atribuir seus valores[2].

Segue um exemplo de código em NuSMV:

```
MODULE main
VAR
request : boolean;
status : {ready,busy};
ASSIGN
init(status) := ready;
next(status) := case
request : busy;
1 : {ready,busy};
esac;
LTLSPEC
G(request -> F status=busy)
```

O programa descrito possui duas variáveis *request* do tipo booleana e status do tipo enumerada *ready*(pronto) ou *busy*(ocupado). Onde 0 denota falso e 1 denota verdadeiro, os valores iniciais e subsequentes da variável *request* são definidos pelo ambiente externo durante a execução do programa. Inicialmente status está *ready*(pronto) e se tornara *busy*(bloqueado) sempre que receber um *request*(requisição) verdadeira. Se a requisição for falsa o próximo valor não será determinado, a verificação da expressão formada no case é feita de cima para baixo, e a primeira verificação será efetuado do lado esquerdo do sinal ':'(dois pontos) que se tiver valor verdadeiro atribuirá seu valor o valor do lado direito para a variável declarada ou portanto se *request* for verdadeiro status receberá *busy*. O valor um está definido como verificação padrão caso nenhuma das anteriores sejam verdadeiras[2].

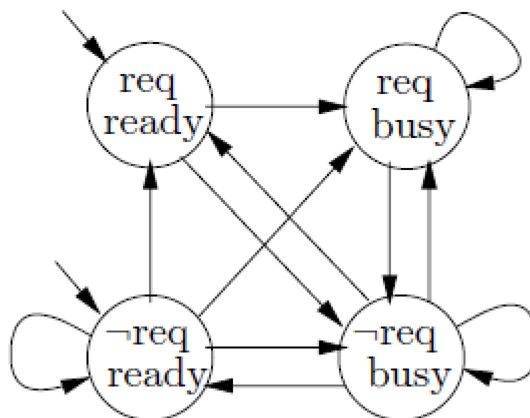

Figura 2 – Modelo de Transições possíveis do programa [2]

A Figura 2 mostra o sistema de transição do programa descrito, onde possuem 4 estados, que estão definidos pelos valores possíveis das duas variáveis, o qual "¬req" significa que o *request* é falso e "req" que é verdadeiro, o programa e seu sistema de transição são não-determinísticos isto é o próximo estado não será unicamente definido, gerando mais de uma possibilidade de próximo estado. As transições dos estados são baseadas na variável *status* que sempre vem em pares e o resultado é a sua transição para um estado sucessor onde *request* é falso ou verdadeiro, nota-se na Figura 2 que a partir do estado ¬req,busy o sistema pode caminhar para quatro estados destinos, ele mesmo e outros três estados. As especificações LTL são introduzidas pela palavra LTLSPEC e são formulas LTL simples[2].

2.4.2 Spin

Spin é uma ferramenta desenvolvida por Gerard J. Holzmann, ele é designada para simulação e verificação de algoritmos, sistemas desenvolvidos no Spin são desenvolvido na linguagem PROMELA que é a sua linguagem específica, ele opera em dois modos o primeiro permite que o usuário se familiarize com o comportamento do seu sistema através

de simulação, o segundo é controlado por uma busca exaustiva de estados que satisfaça alguma propriedade descrita em Lógica temporal Linear[3].

A base teórica do Spin se origina a partir do modelo de comunicação de autômatos por canais delimitados, ele não permite estudar sistema de estados infinitos, porem possui uma disponibilidade de métodos para a redução de estados. A descrição de um sistema programado em PROMELA começa com a definição de constantes e declarações de variáveis globais, as variáveis globais poderão ser acessadas por todos os processos o que farão delas a memória compartilhada do sistema[3].

Considerando um pequeno elevador de três andares seguem as declarações iniciais:

```
bit doorisopen[3];
chan openclosedoor=[0] of {byte,bit};

proctype door(byte i){
do
:: openclosedoor?eval(i),1;
doorisopen[i-1]=1;
doorisopen[i-1]=0;
openclosedoor!i,0
od
}
protoctype elevator(){
show byte floor = 1;

do
:: (floor !=3) -> floor++;
:: (floor !=1) -> floor--;
:: openclosedoor!floor,1;
   openclosedoor?eval(floor),0
od
}

init{
atomic{
run door(1); run door(2); run door(3);
run elevator()}
}
```

O *array* de *bits* doorisopen[3] indica o elevador em cada andar e indica se a porta estará aberta ou não, se o valor for 1 estará aberta e se for 0 estará fechada. Openclosedoor do tipo *chan* fornece a comunicação entre o elevador e a porta do piso, esta comunicação é utilizada para enviar uma mensagem, a mensagem contem o *byte* que é o número do andar que é aplicada a operação e o *bit* que é o pedido enviado para a porta abrir ou fechar. Cada processo é iniciado com a palavra protoctype seguido pelo nome e sua lista de parâmetros que possivelmente pode estar vazia, o processo door aguarda o comando abrir, sinaliza que a porta está aberta, depois de aberta ela é fechada e é sinalizada como fechada, também está descrito um loop infinito *do* em que é feita uma escolha não determinística, suas sequencias de instruções começam a partir do comando "::". O processo elevator contém três comandos de verificação (floor !=3) que verifica se o elevador não está no último andar se não estiver ele acrescenta um andar para o elevador,(floor !=1) verifica se não está no primeiro andar se não estiver ele decrementa um andar para o elevador, a terceira operação envia uma mensagem para o elevador abrir a porta no andar atual e em seguida fechá-la. O passo seguinte consistem em juntar todas essas peças e iniciar a execução do sistema que sempre começa com um *init* que é um processo de iniciação, são executadas as instruções nas três portas e então é executado o processo elevador [3].

As fórmulas a serem verificadas no Spin são LTL e são inseridas em seu interpretador durante a simulação, um exemplo é a fórmula G(open1 $\rightarrow$ X closed1), esta pode ser descrita no interpretador Spin da seguinte forma:

```
#define open1 doorisopen[0]
#define closed1 doorisopen[0]
[] (open1 -> X closed1)
```

O spin então sempre verificará se esta formula será satisfeita ou não em sua execução.

2.4.3 Uppaal

Uppaal é uma ferramenta integrada para modelar, simular e verificar sistemas em tempo real, desenvolvida pela a Universidade de Aalborg e a Universidade de Uppsala [3]. Uppaal permite analisar uma rede de autômatos temporizados com sincronização binaria, ele possui três partes. Em primeiro lugar os processos são descritos com a utilização de um editor gráfico, segundo os sistemas podem ser simulados, é possível optar em realizar um sequencia de instruções e olhar o comportamento do sistema projetado, este comportamento pode ser observado por um simulador gráfico, e finalmente as propriedades podem ser verificadas [3].

Sistemas desenvolvidos em Uppaal também podem ser descritos textualmente como segue o exemplo da Figura 3. Sistemas desenvolvido textualmente em Uppaal se

```
chan app,exit,GoDown, GoUp; //canais

process Train{
clock x;
state far, near{x<30}, on{x<20};
init far;
trans near -> on{
guard x>20, x<30;
assign x:=0;
},
far -> near {
sync App!;
assign x:=0;
},
on -> far {
guard x>10, x<20;
sync Exit!;
};
}
```

Figura 3 – Exemplo de código em Uppaal [3]

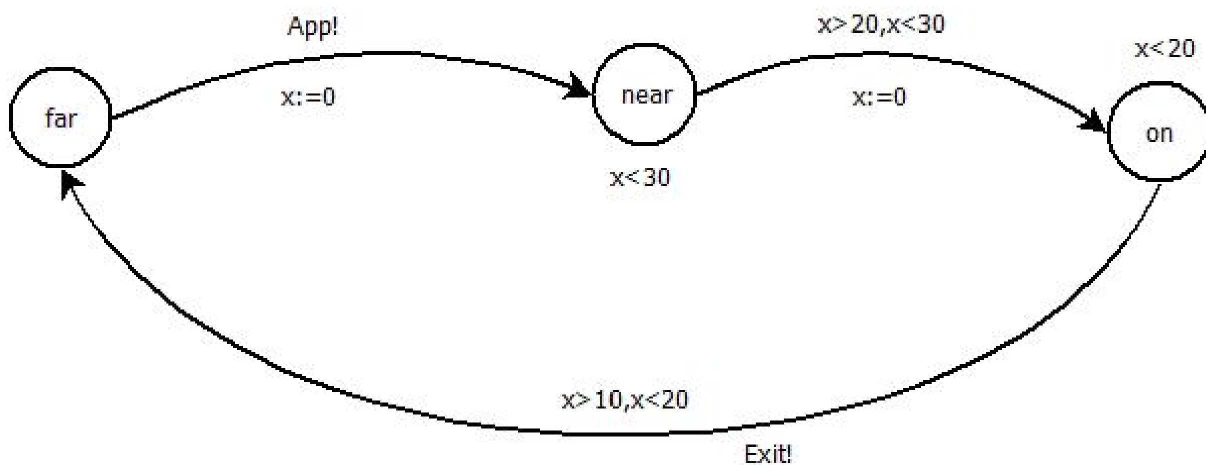

Figura 4 – Modelo Grafico do Train em Uppaal [3]

iniciam com a declaração de variaveis globais como *clocks*, inteiros e canais. Como pode ser observado o Uppaal funciona similar a um automato o exemplo da Figura 4 descreve três estados *far*, *near*, *on* onde suas transições dependerão do *clock* x e sempre que for efetuada uma transição o relógio receberá o valor 0 [3].

3 TRABALHOS RELACIONADOS

Neste capítulo estão descritos as pesquisas que estão relacionados de uma maneira próxima ao deste trabalho, inicialmente é descrito uma verificação formal de politicas de configuração de Firewall, posteriormente uma técnica para a verificação de configurações Firewall baseada em decisões binarias.

3.1 Verificação Formal aplicado a políticas de configuração de Firewall

As configurações do Firewall são cruciais para garantir uma segurança na rede, essas configurações são um conjunto de regras ou política que especificam as propriedades do que será aceito ou descartado pelo Firewall. Essas configurações podem ser complexas alem de serem um grande número de configurações, com isso sempre se tem um interesse em criar métodos computacionais para a verificação, um método de verificação que pode ser utilizado é a Verificação Formal conhecido também como *Model Checking*[4].

Foram definidos dois problemas a serem verificados:

- Acessibilidade: para cada regra r da política, existe um pacote p que faz com que a regra r seja inacessível pelo Firewall (provavelmente está mal configurado).
- Ciclicidade: existe um pacote p, que faz com que o Firewall entre em um ciclo sem aceitar ou rejeitar (certamente mal configurado).

O modelo para verificação consiste uma modelagem atômica do pacote e do estado do Firewall denotado por $M = (\Sigma, \phi, \sigma)$, onde Σ é um número finito de estados, ϕ são as proposições atômicas e σ é a função de validação $\sigma : \Sigma \times \phi \rightarrow Boolean$.

O Modelo pode ser descrito da seguinte forma:

- Estados são escritos como $\{f1 = v1, ...fn = vn\}$,
- Propriedades atômicas são igualdades da forma $f = v$,
- Validação $k(r, f = v)$ é verdade sempre que $(f = v)$ está em r.

Exemplo de modelos atômicos:

- Para pacotes $\{dst0 = 192, dst1 = 168, dst2 = 1, dst3 = 1, dport = 80, \dots\}$
- Para estados do Firewall $\{chain = forward, rule = 0, oiface = eth0, \dots\}$

Podem ser formados produtos de modelos atômicos $M = M1$ x $M2$, onde $E = E1$ x $E2$, $J = J1 + J2$, e $K(x1, x2, j) = Ki(xi)$ quando j está em Ji.

As proposições atômicas do modelo $B(M)$ pode ser dado pela gramatica:

$$A, B, C ::= true \mid false \mid A \wedge B \mid A \vee B \mid \neg A \mid \phi$$

A proposição atômica ϕ de M se estende a função de validação $\sigma : \sum$ x $\phi \rightarrow$ *Boolean* e se escreve $x \models A$ sempre que $\sigma(x, A) =$ true.

O modelo de verificação de política de configuração do Firewall consiste em um modelo $C = (P, S, \gamma, \delta)$, onde P consiste em um modelo de pacote, S em um modelo do estado do Firewall, γ em uma função de decisão γ: $S \rightarrow B(P$ x $S)$ e a relação de transição $\delta \subseteq S$ x Boolean x S. Então pode-se escrever $p \models s \rightarrow t$ sempre que $(s, \sigma(p, s\gamma(s)), t) \in \delta$.

Um modelo atômico enraizado é um modelo M juntamente com uma formula Boolean init $\in B(M)$. Sendo assim segue exemplo de uma configuração Firewall e seu respectivo modelo.

3.2 Fireman uma Ferramenta para Análise e Modelagem de Firewall

A ferramenta Fireman é um framework de análise de firewall, sua análise é aplicada em duas etapas, primeiramente ele análise as regras de uma forma compacta baseado na semântica operacional de um firewall, em seguida suas ACLs são traduzidas em um gráfico de regras, as quais são verificadas se possuem configurações errôneas com base no gráfico gerado. Baseado nas configurações de rede é realizada uma análise para verificar todos os caminhos possíveis das regras que os pacotes podem passar, para firewalls simples não existe possibilidade de ramificação o gráfico da regra é a própria lista, para firewalls que utilizam uma cadeia complexa é gerado um diagrama de todos os seus caminhos possíveis como mostra o exemplo da Figura 5.

O Fireman inicialmente realiza a verificação das ACLs individuais do firewall sem considerar interações com outros Firewalls na rede, após verificar firewall local se não for encontrado erros de configurações ele irá realizar a verificação distribuídas com os demais firewalls da rede com base em seu diagrama gerado fazendo uma checagem do topo para baixo em busa de suas configurações errôneas. Todas essas verificações do fireman são baseadas em BDD(binary decision diagrams) diagramas de decições binarias.

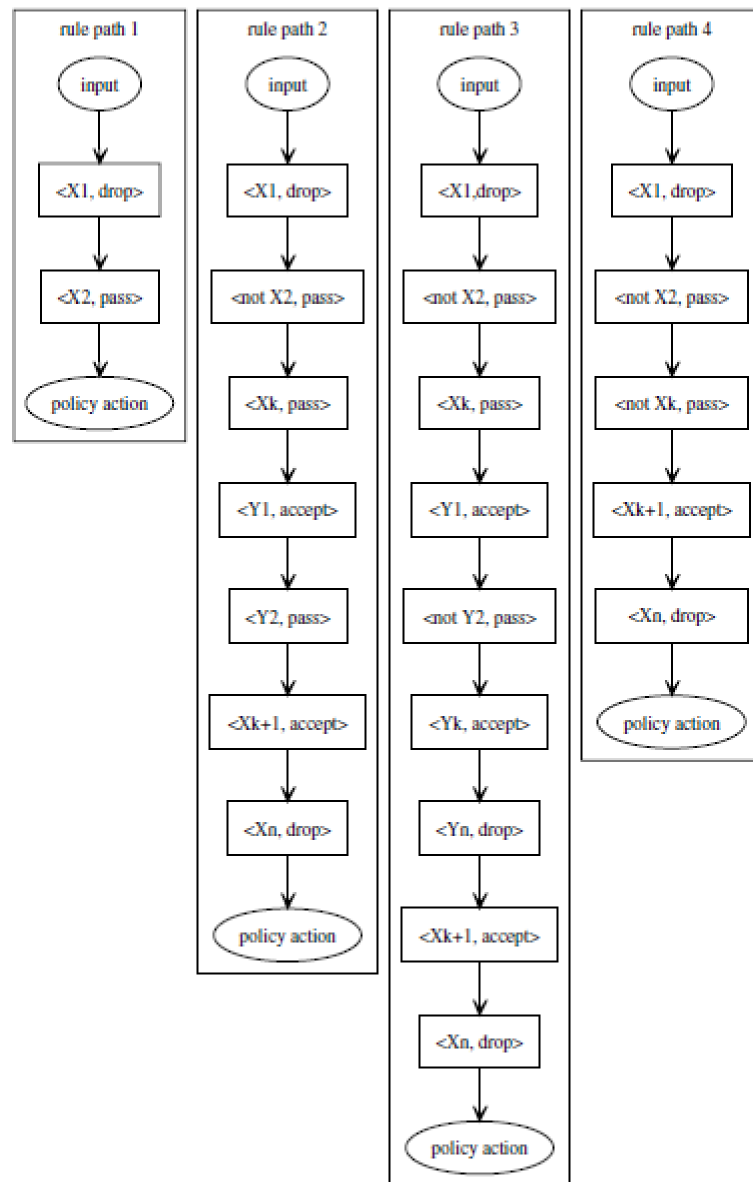

Figura 5 – Cadeia Complexa [5]

4 ESTRUTURAÇÃO DA PESQUISA

Neste capítulo serão retratadas as técnicas utilizadas neste trabalho, assim como a descrição das etapas necessárias para alcançar os objetivos deste trabalho. Este caracteriza-se como uma pesquisa aplicada, pois a pesquisa aplicada tem como objetivo produzir conhecimentos para a aplicação prática dirigido a problemas específicos[14]. Para melhor organização esta pesquisa foi dividida em quatro etapas que serão descritas nas seções a seguir:

- **Fase preliminar:** Nesta fase será definida o campo de pesquisa e também a formulação do problema a ser resolvido bem como, os objetivos do trabalho e como este trabalho poderá ajudar no âmbito dos administradores de *proxy*.
- **Revisão:** Esta fase consiste numa revisão geral das bibliografias relacionadas com o conteúdo de *Model Checking* e também elaborar uma relação de pesquisas que possuem alguma relação com esta.
- **Desenvolvimento:** O desenvolvimento é a parte que inclui todas as etapas anteriores, para alcançar os objetivos propostos. Para isso, o trabalho é dividido em duas partes, conforme segue:
    1. **Modelo do *proxy*** : O objetivo deste trabalho é apresentar um modelo que possa expressar as configurações do *proxy* de forma que elas sejam verificadas formalmente.
    2. **Ferramenta**: Para atingir o objetivo principal deste trabalho é necessário efetuar o desenvolvimento de uma aplicação baseado no modelo desenvolvido, de forma que a verificação das propriedades do *proxy* sejam verificadas de forma automática.
- **Conclusão:** Por fim, realizar uma análise sobre o trabalho, apontando os principais pontos da pesquisa, pontos positivos e negativos, e também discutir possíveis trabalhos para serem elaborados futuramente.

5 MÉTODO PROPOSTO

Nos Capítulos anteriores são apresentados alguns métodos de verificação de configurações de firewall, porém não se tem conhecimento de trabalhos relacionados com verificações de configurações de *proxy*. Assim sendo este Capítulo propõe um método para verificação formal de configurações de *proxy* Squid. Este método utiliza os conceitos de verificação formal conforme descrito no Capitulo 2 que se compõe de três passos: definição do modelo, definição de propriedades a serem satisfeitas e verificação das propriedades no modelo, este modelo também se baseia nos trabalhos citados no Capítulo 3 os quais tratam de verificação de regras de *firewall*.

O método proposto para verificar formalmente as regras de *proxy* consiste nas etapas a seguir:

1. Definição de um modelo para expressar as configurações de um *proxy* e o modo de construção deste modelo;
2. Um modo de definir quais propriedades são desejáveis no modelo determinado;
3. Validação dessas propriedades definidas perante o modelo proposto.

5.1 Modelo do Proxy

A intuição utilizada para modelar as regras de *proxy* consiste em representar como ocorre a sua aplicação em um servidor real. O modelo proposto consiste em um autômato construído a partir das regras do *proxy*. Está construção é feita a partir das linhas que contem os comandos `acl` e `http_access`, a partir desses comandos é que serão geradas os estados do autômato e suas marcações. Com a construção completa do autômato são obtidos todos os caminhos possíveis que as requisições ao *proxy* podem percorrer.

A definição do autômato $\mathcal{M}$ que modela uma lista $\mathcal{H}$ que contem todos os comandos `http_access` do arquivo de configuração é definido por:

$$\mathcal{M}(\mathcal{H}) = \langle Q, q_0, \mathcal{F}, \mathcal{B}, \delta \rangle \tag{5.1}$$

onde:

- Q: é o conjunto de estados do modelo;
- q_0 é o estado inicial;
- $\mathcal{F} = \{allow, deny\}$: é um conjunto de estados finais especiais;

- $\mathcal{B} = \{True, False\}$: alfabeto de transição;
- $\delta : Q \times \mathcal{B} \rightarrow Q \cup F$: é a função de transição dos estados;
- $\alpha : Q \rightarrow 2^{ACL}$: é a função de rotulação dos estados.

A construção do autômato $\mathcal{M}$ a partir de uma lista $\mathcal{H}$ é feita para cada linha da lista, onde é gerado um novo estado $q \in Q$. A construção é feita desta maneira pois conforme ja citado no Capítulo 2 as regras do Squid são interpretadas de cima para baixo. Uma linha da lista $\mathcal{H}$ também contém as informações de qual será estado final se este estado for satisfeito perante a entrada e as acl's que serão rotuladas neste estado. As rotulações de cada estado são atribuídas a partir da função $F_{acl}(h)$ onde $h \in \mathcal{H}$ esta função retorna todas acl's contidas em h. O estado final contido na linha h pode ser obtido pela função $F_{access}(h)$. As transições são geradas a partida da função δ que dado um estado $q \in Q$ e um booleano $b \in \mathcal{B}$ o próximo estado poderá ser um estado final $f \in \mathcal{F}$ ou um estado $q \in Q$. A construção do autômato pode ser observada no Algoritmo 1.

Algoritmo 1: $constru ir Modelo(\mathcal{H})$

```
input  : H
output: autômatoM(H)
begin
    for h ∈ H do
        F_http(q) ← h;
        Q ← Q ∪ q;
    end
    for q ∈ Q do
        if F_http(q) = h_0 then
            q_0 ← q;
        h ← F_http(q);
        α(q) ← F_acl(h);
        δ(q, True) ← F_access(h);
        if ∃q' then
            δ(q, False) ← q';
        else
            δ(q, False) ← F − {F_access(h)};
        end
    end
    return M(H);
end
```

Além das funções ja citadas o Algoritmo 1 também faz o uso da função $F_{http}(q)$ função que recebe ou retorna o índice de um $q \in Q$.

O Algoritmo 1 começa recebendo uma lista $\mathcal{H}$ e constrói o autômato da seguinte forma:

1. Para todo $h \in \mathcal{H}$ será adicionado um novo estado q ao conjunto de estados que Q;
2. Para todo $q \in Q$ percorrera todos os estados do autômato.
   a) Se $F_{http}(q) = h_0|$ então será definido o estado inicial do automato através a função $F_{http}(q)$ que retorna o índice de um $h \in \mathcal{H}$ que se for o primeiro então atribuirá o valor do estado atual ao q_0;
   b) Atribui-se o índice de um estado q a um h através da função $f_{http}(q)$ que retorna um $h \in \mathcal{H}$ para que h possa ser utilizado nos próximas decisões do algoritmo;
   c) $\alpha(q) \leftarrow F_{acl}(h)$ adiciona as rotulações do estado q;
   d) $\delta(q, True) = F_{access}(h)$ Atribui a transição caso a transição seja verdadeira para um estado final obtido pela função $F_{access}(h)$;
   e) Se $\exists q'$ então a transição False para o próximo estado será q'
   f) Caso contrário o próximo estado a transição False será $\delta(q, False) = \mathcal{F} - \{F_{access}(h)\}$ o que retorna o inverso da transição True;
3. Retorna o Autômato $\mathcal{M}(\mathcal{H})$ construído.

5.2 Definição das Características do *Proxy*

As regras de *proxy* podem ser representadas formalmente por um autômato $\mathcal{M}(\mathcal{H})$. Para avaliar as configurações expressas neste autômato é preciso definir as propriedades que o modelo pode satisfazer além do formato destas propriedades. As propriedades do modelo proposto são avaliadas por meio de caminhos (ou *traces*) que consistem em uma sequência finita de estados. Considerando que um caminho π é uma sequencia finita $\pi = \{\pi_0, \pi_1, ...\pi_n | \pi_1, ...\pi_{n-1} \in Q \wedge \pi_n \in f\}$. Uma propriedade a ser satisfeita é uma proposição e tem o formato $\varphi(Req, \mathcal{F})$ onde $Req = (SRC, DST)$ e `SRC` representa o ip do solicitante e `DST` representa o destino solicitado e $\mathcal{F}$ o estado final que pretende-se alcançar. Uma proposição φ é satisfeita no modelo $\mathcal{M}$ se existe um caminho π tal que $\mathcal{M} \models \varphi$ esta relação de satisfação consiste em verificar se existe um caminho π tal que π_n seja igual $\mathcal{F} \in \varphi$. A Relação de satisfação pode ser observada na Figura 6.

As definições (5.2) e (5.3) indicam proposição satisfeita e não satisfeita, respectivamente. Alguns conectores da lógica proposicional são definidos nas propriedades (5.4) a (5.6).

A simulação das propriedades no autômato pode ser observada no Algoritmo 2 onde a entrada é um estado q e uma propriedade φ. Também na execução da simulação

$$\mathcal{M}, \pi \models \varphi \Leftrightarrow \exists \pi \in \mathcal{M} | \pi_n = f \in \varphi \tag{5.2}$$
$$\mathcal{M}, \pi \not\models \varphi \Leftrightarrow \exists \pi \in \mathcal{M} | \pi_n = f \notin \varphi \tag{5.3}$$
$$\mathcal{M}, \pi \models \varphi \wedge \varphi' \Leftrightarrow \mathcal{M}, \pi \models \varphi \text{ e } \mathcal{M}, \pi \models \varphi' \tag{5.4}$$
$$\mathcal{M}, \pi \models \varphi \vee \varphi' \Leftrightarrow \mathcal{M}, \pi \models \varphi \text{ ou } \mathcal{M}, \pi \models \varphi' \tag{5.5}$$
$$\mathcal{M}, \pi \models \neg\varphi \Leftrightarrow \mathcal{M}, \pi \not\models \varphi \tag{5.6}$$

Figura 6 – Semântica das Propriedades.

pode-se observar a função auxiliar F_{final} a qual retorna o estado final que foi definido na propriedade que é o estado pretende se atingir. Enquanto o estado atual não for um estado final(`allow ou deny`) a função é chamada recursivamente, nesta chamada é utilizado como parâmetro a função de transição δ que retorna o próximo estado. A função δ também utiliza o Algoritmo 3 que é utilizado para validar se as propriedades são verdadeira no estado que se encontra, isso definirá se a transição será True ou False.

Algoritmo 2: *simulaAutomato*(q, φ)

input : um estado $q \in Q \cup \mathcal{F}$ do autômato$\mathcal{M}(\mathcal{H}), \varphi$
output: Satisfação Satisfaz ou Não Satisfaz
begin
  if $q \in \mathcal{F}$ **then**
    if $q = F_{final}(\varphi)$ **then**
      return Satisfaz;
    else
      return Não Satisfaz;
    end
  else
    return $simulaAutomato(\delta(q, match(f_\alpha(q), \varphi), \varphi)$
  end
end

1. A função recursiva simulaAutomato faz a simulação do automato a partir de um estado q e uma entrada φ;

2. Se $q \in \mathcal{F}$ se q pertence aos estados finais $\mathcal{F}$ então:

   a) Se $q = F_{final}(\varphi)$ então o estado final do simulação é igual o estado final da entrada φ então retorna Satisfaz que significa que φ foi satisfeita no automato;

   b) Caso contrário retorna Não Satisfaz;

3. Caso contrário a função simulaAutomato é chamada recursivamente com o próximo estado obtido pela transição δ e pela função `match` que retorna um valor true ou false;

Como já citado o Squid *proxy* percorre suas regras até que seja encontrada uma combinação(match), então após a combinação é definido se será permito ou não o acesso solicitado(`allow ou deny`). O Algoritmo 3 descreve os passos para a verificação de uma propriedade φ em um estado $q \in Q$ do autômato. A entrada do algoritmo recebe a função auxiliar $F_\alpha(q)$ e uma propriedade φ, a função $F_\alpha(q)$ retorna as acl's presentes no estado q que então são verificadas de acordo com a propriedade φ para determinar se existe uma combinação(match) ou não. Está função é que será responsável em efetuar o retorno de um boolean do alfabeto de transição, transição que ocorre com o retorno para o Algoritmo 2.

Algoritmo 3: $mathc(F_\alpha(q), \varphi)$

input : $F_\alpha(q), \varphi$
output: true ou false
begin
  boolean aux;
  for $acl \in F_\alpha$ **do**
    if $F_{dst}(\varphi) \notin acl\ \ e\ F_{src}(\varphi) \notin acl$ **then**
      if !acl **then**
        $aux \leftarrow true$;
      else
        return false;
      end
    else
      $aux \leftarrow true$;
    end
  end
  return aux;
end

1. Função `match` que recebe as acl contidas em um estado e um φ e retonar um boolean true ou false;

2. For para verificar todas acls do estado que são obtidas pela função $F_\alpha(q)$

   a) Se dst e src não pertencem a acl então:

      i. Se acl for negada aux recebe true;

      ii. caso contrario retorna false;

   b) Caso contrario aux recebe true;

3. Retorna a variavel auxiliar aux;

5.3 Aplicação do Método

Um exemplo de aplicação do método proposto pode ser observado aplicando-o em um arquivo de configuração simples de uma pequena empresa. Como segue a configuração descrita na Figura 7.

```
acl all src 0.0.0.0/0.0.0.0
acl redelocal src 192.168.0.0/24
acl chefe src 192.168.0.1
acl setorAdm src 192.168.0.2 192.168.0.3 192.168.0.4 192.168.0.5 192.168.0.6
acl funcionarios src 192.168.0.7 192.168.0.8 192.168.0.9 192.168.0.10
acl google dst 200.195.155.223
acl facebook dst 31.13.80.36
acl youtube dst 200.192.155.223
acl noticias dst 186.192.90.5
http_access allow chefe
http_access deny facebook
http_access allow youtube !funcionarios
http_access deny youtube
http_access allow noticias
http_access allow google
http_access deny all
```

Figura 7 – Exemplo de configuração.

A configuração é descrita da seguinte forma (1) o `chefe` tem total permissão no *proxy*, (2) o acesso ao `facebook` é bloqueado, (3) o acesso ao `youtube` é permitido exceto para `funcionarios`, (4) acesso ao `youtube` é bloqueado, (5) acesso a `noticias` é permitido, (6) acesso ao `google` é permitido, (7) se não for encontrado nenhum acesso que corresponde as regras anteriores o acesso é bloqueado. O autômato que representa o arquivo de configuração da Figura 7 é construído através do Algoritmo 1. O autômato construído $\mathcal{M}(\mathcal{H})$ é demonstrado na Figura 8.

Após a construção do autômato as simulações de propriedades podem ser executadas, as entradas seguem o padrão proposto na Seção 5.2 onde são definidos os padrões de entradas do autômato. Com as entradas definidas são executadas as simulações através do Algoritmo 2, onde são encontrados estados que satisfazem a entrada através do Algoritmo 3 que verifica se a entrada foi satisfeita no estado atual da execução.

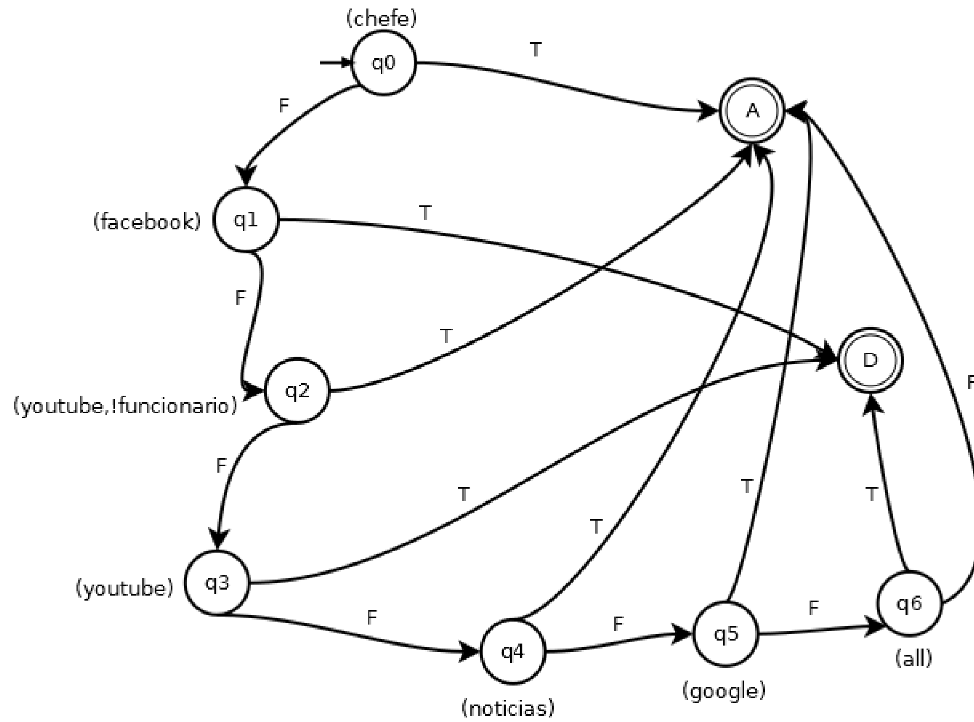

Figura 8 – Modelo

Se um `match` for encontrado a execução procede para o estado final onde é verificado a satisfação de acordo com o estado pretendido na entrada, então é feito o retorno de satisfação ou não satisfação da propriedade. A Tabela 1 apresenta algumas simulações que contém as seguintes informações: entrada do automato, o caminho percorrido, e a relação de satisfação. .

Tabela 1 – Entradas de teste

	Entrada			Caminho	Final
1	192.168.0.1	200.192.90.5	allow	q_0,A	Satisfaz
2	192.168.0.7	192.168.0.7	allow	q_0, q_1, q_3,D	Não Satisfaz
3	192.168.0.3	200.192.155.223	allow	$q_0, q_1, q_2, q_3, q_4, q_5$,A	Satisfaz
4	192.168.0.4	200.56.12.8	deny	$q_0, q_1, q_2, q_3, q_4, q_5, q_6$,D	Satisfaz

6 FERRAMENTA DE VERIFICAÇÃO

O método para a verificação das regras do *proxy* Squid foi elaborado através dos conceitos de orientação a objetos, pois conforme descrito no Capítulo 5, o modelo consiste em um autômato que é composto por seis elementos. Para auxiliar no planejamento do desenvolvimento da ferramenta foi utilizado a linguagem UML [15] ilustrada pela ferramenta Astah *Professional*[16] e para o desenvolvimento foi escolhida a linguagem JAVA [17], por se tratar de uma linguagem multiplataforma. O funcionamento da ferramenta é apresentado na Figura 9. Os retângulos inclinados indicam a entrada e saída da ferramenta, os retângulos são o módulos da ferramenta e as setas indicam a comunicação entre os módulos.

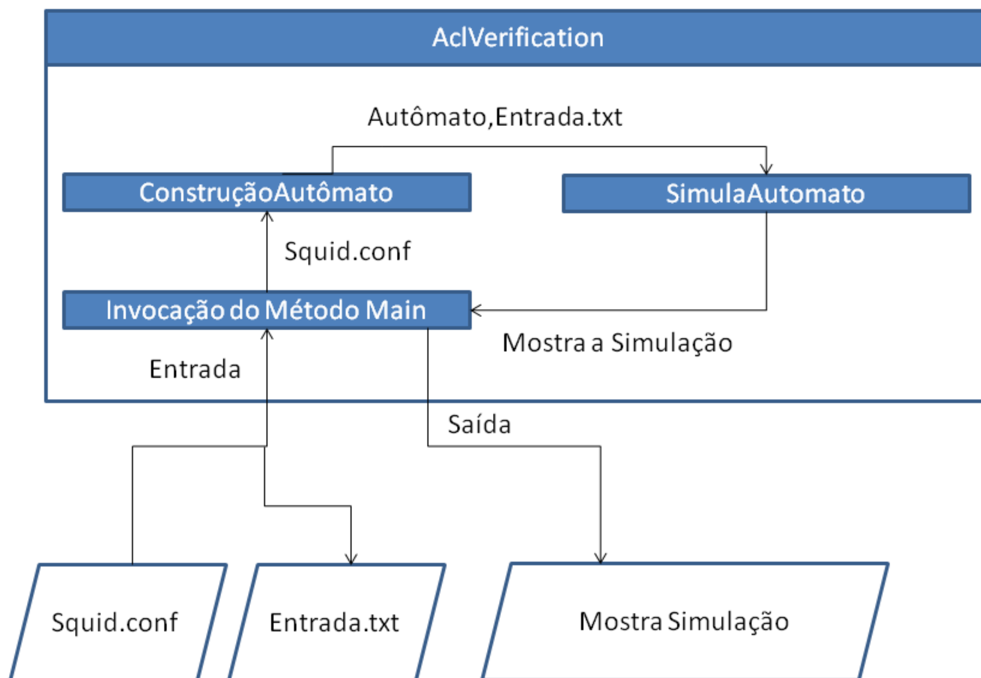

Figura 9 – Funcionamento da ferramenta

A ferramenta possui uma classe principal `Main` que ao ser invocada precisa receber o arquivo de configurações do *proxy* e as entradas a serem testadas e retorna na tela o resultado da simulação das entradas. O módulo `ConstruçãoAutômato` constrói o autômato a partir das configurações do *proxy* em seguida o autômato e a entrada são enviados para o módulo `SimulaAutômato`. O módulo SimulaAutômato realiza a verificação das

propriedades que foram definidas no arquivo de entrada conforme a Seção 5.2 e mostra na tela se as mesma estão satisfeitas ou não perante o autômato.

6.1 Análise e Projeto

Para o planejamento e analise do projeto foi utilizado como suporte os diagramas da UML[15]. O diagrama da Figura 10 descreve os pacotes da ferramenta. A leitura do arquivo de configuração do *proxy* é feita no pacote `parser`. No pacote `algoritmos` contém as classes `construirAutomato`, `SimularAutomato` e `entrada` que por sua vez são responsáveis por construir e efetuar a simulação do automato. A o pacote `modelo` contem as classes `Acls` e `HttpAccess` que estão relacionadas com a construção do autômato. Já o pacote `Automato` contém as classes `Automato`, `Entrada` e `Estado` que são os componentes do autômato. O pacote `inparser` é responsável para interpretar as entradas a serem testadas. E por fim o pacote `main` que ao ser invocado é responsável pela execução dos algoritmos para verificação.

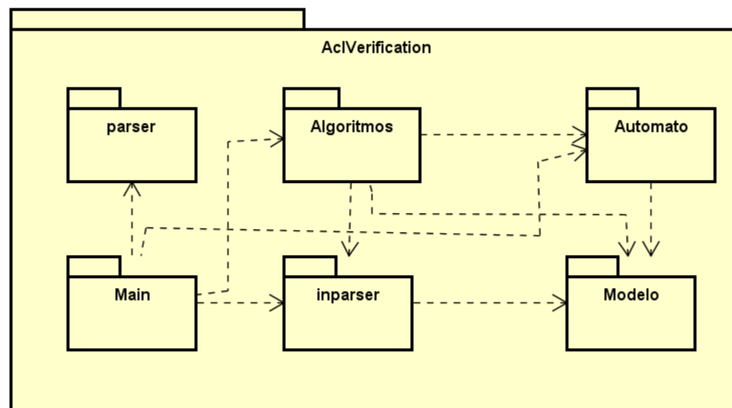

Figura 10 – Diagrama de pacotes da ferramenta

Com a estrutura da ferramenta definida no diagrama de pacotes, as demais subseções descrevem detalhadamente o funcionamento da ferramenta.

6.1.1 Construção do Autômato

Para que o Autômato seja construído corretamente de acordo com seu arquivo de configuração foi desenvolvido um analisador sintático. O analisador sintático é capaz de interpretar o arquivo e gerar as informações necessárias para a construção do autômato. O funcionamento do analisador sintático pode ser observado no diagrama da Figura 11 Ao interpretar o arquivo de configuração do *proxy* o analisador sintático retorna dois tipos de objetos, um objeto do tipo `Acl` que retorna uma acl do arquivo de configuração e um objeto do tipo `HttpAccess` que retorna uma linha de permissão de acesso do arquivo de configuração. Até que o arquivo de configuração chegue ao fim o analisador sintático

adiciona ambos objetos a duas listas uma lista de `Acl` e outra lista de `HttpAccess`, ambas que serão utilizadas posteriormente para a construção do autômato.

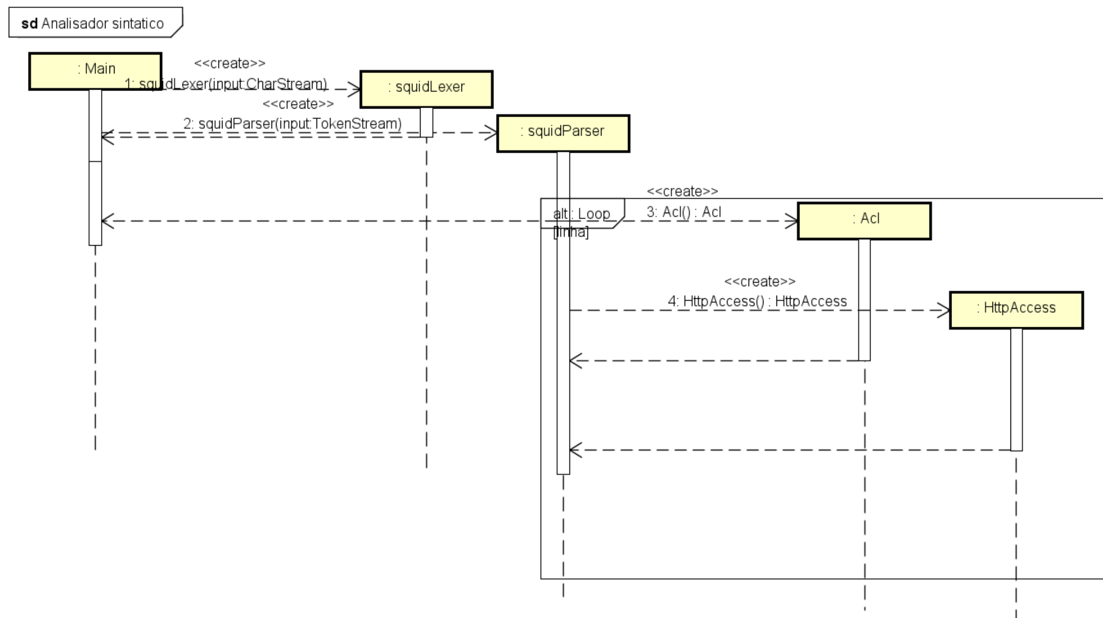

Figura 11 – Diagrama de sequencia do analisador sintático

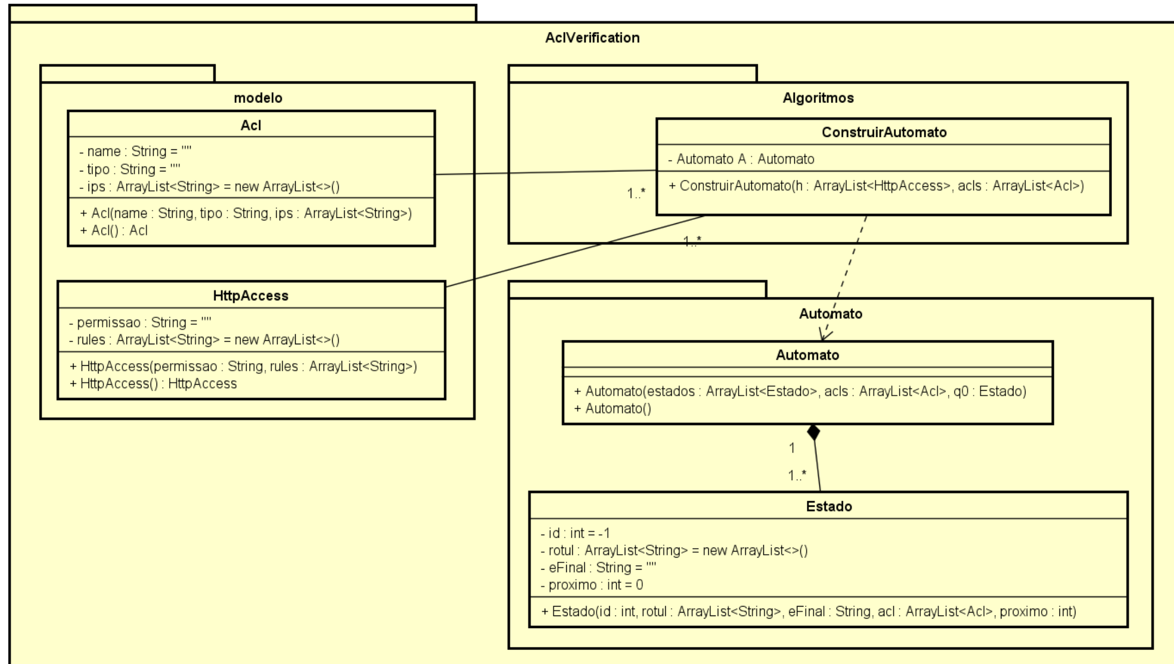

Figura 12 – Classes de representação e construção do autômato

A construção do automato é representada na Figura 12, onde as classes representam um autômato e suas funcionalidades de construção. A classe `ConstruirAutomato` possui um construtor que recebe duas listas de objetos uma de `Acl` e outra de `HttpAccess` e efetua a construção do autômato. Os estados do autômato são gerados a partir da lista

dos objetos `HttpAccess`, sendo criado um estado para cada objeto, cada estado com suas marcações(Acl's), seu estado final e o estado seguinte.

Com a construção do autômato completa ele está pronto para a simulação que será descrita na próxima seção.

6.1.2 Simulação do Autômato

Para que o autômato gerado seja válido será necessário efetuar a sua simulação. Antes de realizar a simulação é preciso definir as propriedades a serem verificadas conforme foi descrito na Seção 5.2. A classe `SimulaAutomato` é responsável pela simulação seus respectivos componentes estão representados na Figura 13. Ao ser instanciada a classe define em seu construtor o autômato que será simulado. A simulação do autômato é iniciada com a invocação da função `Satisfaz(entrada)` que recebe por parâmetro uma entrada para verificação, após receber a entrada a função efetua recursivamente a chamada da função `match(estado,entrada)` até que encontre uma combinação coma entrada. A função `match` tem o intuito de encontrar algum estado no autômato que combine com a entrada informada, com a localização deste estado é efetuada a relação de satisfação ou não satisfação da propriedade, com a comparação dos estados finais de entrada e do estado final encontrado na simulação.

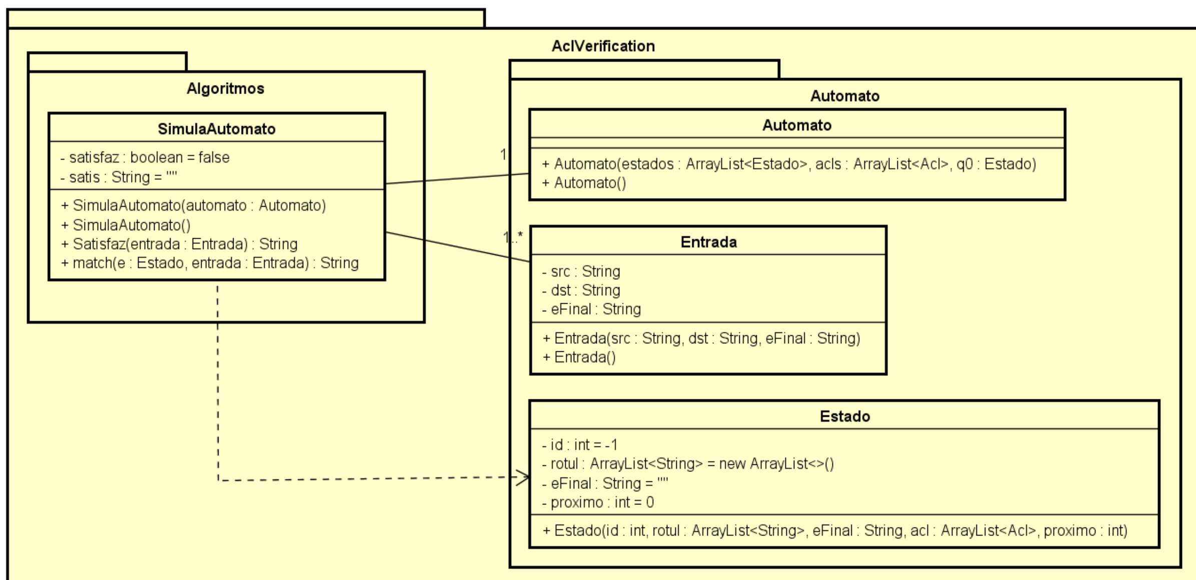

Figura 13 – Classes de representação da simulação do autômato

6.2 Desenvolvimento

Conforme a análise e projeto da ferramenta, descrito na seção anterior, a implementação da ferramenta foi feita com o uso da Linguagem Java[18]. Por se tratar de uma

linguagem orientada a objetos o que permite uma melhor organização do código, além de permitir a utilização de componentes e bibliotecas[17]. Para auxiliar na construção da ferramenta também foi utilizado um ambiente integrado de desenvolvimento, NetBeans IDE [19], que forneceu auxilio para a compilação, construção, e verificação do código da ferramenta. No auxilio da construção da ferramenta também foram utilizadas duas bibliotecas de código aberto sobe a licença BSD [20]. Com a finalidade de interpretar o arquivo de configuração do *proxy* foi utilizado a biblioteca ANTLR[21] que foi responsável por gerar os analisadores léxico e sintático em código Java, permitindo obter as informações do arquivo de configuração, necessárias para a construção do autômato. Na implementação da simulação foi utilizado a biblioteca *IP/Subnet Calculator* [22], utilizada para efetuar a verificação se o IP de entrada encontra-se dentro de um domínio de rede da regra do *proxy*, assim validar a entrada em determinado estado do autômato. Ao invocar a classe main(método principal) são necessários passar os parâmetro da localização do arquivo de configuração e de um arquivo com as entradas a serem simuladas no autômato, após a invocação o resultado das simulações são exibidos na tela.

Para a construção do autômato a partir do arquivo de configuração do *proxy*, foi elaborado uma gramatica na ferramenta ANTLR [21] que gera o código-fonte em Java dos analisadores léxico e sintático. Os quais são responsáveis por obter os objetos `Acl` e `HttpAccess` a partir do arquivo de configuração. Após percorrer todo o arquivo de configuração o analisador terá uma lista de todos objetos `Acl` e `HttpAccess`, estas lista serão utilizadas para a construção do autômato.

A construção do autômato que representa as configurações é realizada conforme o código da Figura 14, que implementa a construção do autômato. Na linha (4) é criado uma lista de estado que serão preenchidos com os estados do autômato. O `for` da linha(5) é responsável por percorrer a lista de `HttpAccess` e criar um estado para cada `HttpAccess`. A linha (7) é responsável por fazer a busca de cada `Acl` que pertence a cada `HttpAccess`. Com os atributos do estado definido o estado é adicionado a lista conforme a linha (18). Na linha (21) é efetuada a construção do autômato informando sua lista de estado, o estado inicial e as acls que existem no autômato.

A simulação das propriedades no automato é feita através da função `match` da classe `SimulaAutomato` a simulação é feita da seguinte forma:

1. A função `match` recebe um estado e uma entrada para teste;
2. A função percorrer as acls do estado;
3. Verifica se a `acl` é do tipo `src` ou `dst` para comparar com os campos da entrada `src` ou `dst`;
4. Após definir qual tipo de `acl` a função verifica se a entrada satisfaz o estado;

5. Se não for satisfeito a função é chamada recursivamente com o próximo estado;

6. Se for satisfeita a função retorna o estado final que foi encontrado `allow` ou `deny`.

Após o retorno da função `match` são impressos na tela de comando o estado final pretendido e o estado final atingido juntamente com o tempo de execução.

```
1  public ConstruirAutomato( ArrayList<HttpAccess> h, ArrayList<Acl> acls){
2   this.h = h;
3   this.acls = acls;
4   ArrayList<Estado> e = new ArrayList<>();
5   for (int i = 0; i < h.size(); i++) {
6    ArrayList<Acl> acl = new ArrayList<>();
7    for (int j = 0; j < h.get(i).getRules().size(); j++) {
8     String aux = h.get(i).getRules().get(j);
9     char [] k = aux.toCharArray();
10    if ( !Character.isDigit( k[ 0 ] ) ) {
11     for (int l = 0; l < acls.size(); l++) {
12      if(aux.contains(acls.get(l).getName())){
13   acl.add(new Acl(acls.get(l).getName(),acls.get(l).getTipo(),acls.get(l).getIps()));
14                        }
15                    }
16             }
17             }
18            e.add(new Estado(i,h.get(i).getRules(),h.get(i).getPermissao(),acl,i+1));
19         }
20         e.get(e.size()-1).setProximo(-1);
21         Automato b = new Automato(e,acls,e.get(0));
22         this.a=b;
23     }
```

Figura 14 – Código-fonte do construtor *ConstruirAutomato*.

7 ESTUDO DE CASO

O estudo de caso apresentado neste trabalho retrata um rede de um empresa de médio porte, onde os acessos a determinado conteúdo é limitado ao cargo de cada funcionários apresentam dentro da empresa. O arquivo de configuração apresentado basicamente controla acesso a destinos com conteúdo, sobre empregos, vídeos, dispositivos de busca e redes sociais. Também será apresentado como é possível efetuar a verificação das configurações pela ferramenta `AclVerification`.

7.1 Aplicação da Ferramenta

Para que seja efetuada a verificação de um arquivo de configuração na ferramenta `AclVerification` é necessário que seja criado um arquivo onde contenha apenas as configurações do Squid *proxy* relacionadas a `acl` e a`http_access` conforme o exemplo da Figura 15.

```
acl special src 10.1.1.15
acl goodkey dst 242.128.15.2
acl allow_proxy src  10.1.1.30
http_access allow special
http_access allow goodkey
http_access allow allow_proxy
http_access deny all
```

Figura 15 – Exemplo Configuração Squid.

A construção do autômato das configurações da Figura 15 pode ser observada na Figura 16. As propriedades a serem verificadas no arquivo de configuração devem ser descritas em um arquivo de texto da seguinte forma: `src;dst;(allow ou deny)` conforme o exemplos da Figura 17.

Como a configuração do *proxy* e os arquivos de entradas estão definidos de acordo com a ferramenta basta uma invocação do método `main` por linha de comando passando através de parâmetro os endereços do arquivo de configuração e do arquivo de entrada. Após a execução são exibidos na tela de comando da seguinte forma:

```
(estado pretendido;estado alcançado;tempo de execução(nano segundos))
```

o exemplo da execução dos arquivos exemplificados anteriormente é descrito na Figura 18.

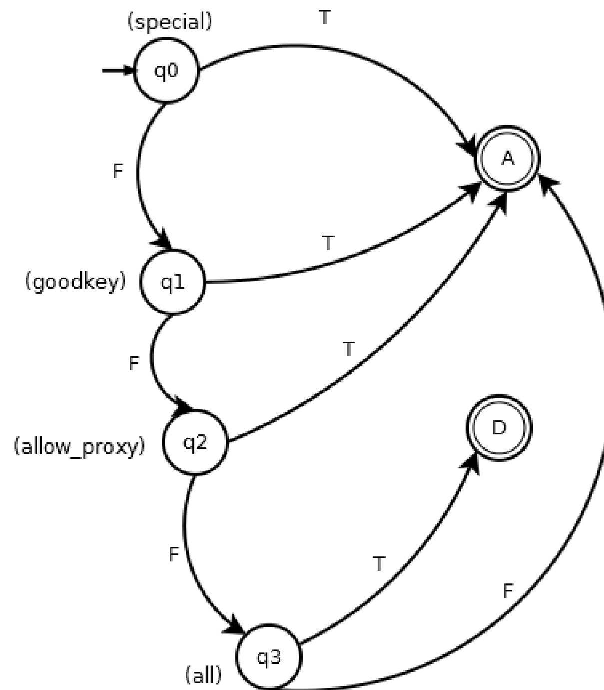

Figura 16 – Exemplo de Autômato construído *ConstruirAutomato*.

```
10.1.1.15;242.128.15.2;allow
10.1.1.30;242.128.15.2;deny
10.1.1.30;242.128.15.2;allow
10.1.1.15;242.128.15.2;deny
```

Figura 17 – Exemplo de entrada para verificação.

```
allow;allow;89259
deny;allow;26746
allow;allow;19282
deny;allow;6843
```

Figura 18 – Exemplo de Saida.

7.1.1 Descrição do Caso

Conforme ja citado nos capítulos anteriores, grande maioria dos *proxys* possuem uma quantidade extensa de regras, o que dificultando para o administrador do *proxy* encontrar o erro.

O arquivo de configuração deste estudo é composto das seguintes regras:

- A empresa possui um grupo de usuários especiais que pertencem a diretoria que tem total acesso ao *proxy*;

- O setor de TI tem acesso total;
- Todos podem acessar o site da empresa;
- O gerente possui total acesso ao *proxy*;
- A redes sociais são bloqueadas para os demais usuários;
- Acesso a tutoriais são permitidos;
- O acesso aos vídeos locais da empresa é permitido;
- Acesso a vídeos externos é bloqueado;
- Todos podem acessar a lista de empregos disponível na empresa;
- Após as demais restrições os acessos a rede são permitidos;
- Caso contrario as restrições anteriores o acesso é bloqueado.

7.1.2 Aplicando a Ferramenta no Caso

O arquivo de configuração que descreve as regras do estudo de caso citado na seção anterior pode ser observado na Figura 19. Para que suas configurações sejam testadas no modelo é necessário gerar as entradas para que as configurações requeridas. As entradas à serem simuladas são expressas na Figura 20.

O resultado da execução da ferramenta `AclVerification` pode ser observada na Tabela 2. Ao observar a tabela nota-se que todos os casos de teste foram satisfeitos, exceto o caso(13), pois trata-se de um caso em que um usuário comum da rede tenta efetuar o acesso aos vídeos. Os casos (1) a (5) representam todos os usuários especias da rede. Os casos (6) a (10) representam os profissionais de TI, já o caso (11) representa o acesso ao site da empresa, o caso (12) representa o acesso do gerente, o caso (13) e (16) representam acesso ao tutorial, o casos (14) e (15) representam o bloqueio as redes sociais, o caso (17) representa o acesso aos vídeos da empresa, o caso (18) representa a proibição de acesso aos vídeos externos, o caso (19) representa o acesso aos empregos da empresa, o caso (20) representa a liberação do acesso a rede local exceto para as demais restrições ja definidas, o caso (21) representa o acesso negado para qualquer outro tipo de requisição ao *proxy*. Neste caso após observar os resultados dos testes, pode-se concluir que o *proxy* antede aos requisitos descritos na Seção 7.1.1.

```
acl all 0.0.0.0/0.0.0.0
acl localnet src 10.1.1.0/24
acl special src 10.1.1.15 10.1.1.16 10.1.1.17 10.1.1.18
acl ticc src 10.1.1.20 10.1.1.21 10.1.1.22 10.1.1.23 10.1.1.24
acl gerente src  10.1.1.30
acl allow_tutorial dst 200.15.26.12
acl allowlocal_movie dst 195.15.60.2
acl allow_jobs dst 201.225.65.32
acl siteempre dst 242.128.15.2
acl proxy dst 201.1125.63.12
acl social dst 31.13.85.36
acl tutorial dst 32.15.75.16
acl movie dst 141.0.174.42
acl jobs dst 208.71.195.142
http_access allow special
http_access allow siteempre
http_access allow ticc
http_access allow gerente
http_access deny proxy
http_access deny social
http_access allow allow_tutorial
http_access allow allowlocal_movie
http_access deny movie
http_access allow allow_jobs
http_access allow localnet
http_access deny all
```

Figura 19 – Arquivo de Configuração da Empresa.

```
//special
10.1.1.15;141.0.174.42;allow
10.1.1.16;141.0.174.42;allow
10.1.1.17;31.13.85.36;allow
10.1.1.18;141.0.174.42;allow
10.1.1.30;31.13.85.36;allow
//ticc
10.1.1.20;141.0.174.42;allow
10.1.1.21;31.13.85.362;allow
10.1.1.22;141.0.174.42;allow
10.1.1.23;141.0.174.42;allow
10.1.1.24;31.13.85.36;allow
//site empre
10.1.1.8;242.128.15.2;allow
//gerente
10.1.1.30;31.13.85.36;allow
//tutorial
10.1.1.60;200.15.26.12;allow
//social bloqueada
10.1.1.5;31.13.85.36;deny
10.1.1.6;31.13.85.36;deny
//tutorial
10.1.1.7;200.15.26.12;allow
//videos empresa
10.1.1.10;195.15.60.2;allow
//videos externos
10.1.1.14;141.0.174.42;deny
//empregos
10.1.1.32;201.225.65.32;allow
//redelocal
10.1.1.34;201.15.23.14;allow
//all
192.168.0.1;201.15.23.14;deny
```

Figura 20 – Casos de Teste.

Tabela 2 – Resultado da Simulação

Caso	Entrada			Saida		Tempo(ns)
1	10.1.1.15	141.0.174.42	allow	allow	allow	49139
2	10.1.1.16	141.0.174.42	allow	allow	allow	9641
3	10.1.1.17	31.13.85.36	allow	allow	allow	9952
4	10.1.1.18	141.0.174.42	allow	allow	allow	8397
5	10.1.1.30	31.13.85.36	allow	allow	allow	34211
6	10.1.1.20	141.0.174.42	allow	allow	allow	33277
7	10.1.1.21	31.13.85.362	allow	allow	allow	27369
8	10.1.1.22	141.0.174.42	allow	allow	allow	40431
9	10.1.1.23	141.0.174.42	allow	allow	allow	31412
10	10.1.1.24	31.13.85.36	allow	allow	allow	47584
11	10.1.1.8	242.128.15.2	allow	allow	allow	18350
12	10.1.1.30	31.13.85.36	allow	allow	allow	39187
13	10.1.1.60	200.15.26.12	allow	allow	allow	2449498
14	10.1.1.5	31.13.85.36	deny	deny	deny	84594
15	10.1.1.6	31.13.85.36	deny	deny	deny	51005
16	10.1.1.7	200.15.26.12	allow	allow	allow	147418
17	10.1.1.10	195.15.60.2	allow	allow	allow	101078
18	10.1.1.14	141.0.174.42	deny	deny	allow	75886
19	10.1.1.32	201.225.65.32	allow	allow	allow	115695
20	10.1.1.34	201.15.23.14	allow	allow	allow	78063
21	192.168.0.1	201.15.23.14	deny	deny	deny	88326

8 CONCLUSÃO

Este trabalho apresentou, um método para lidar com a verificação de configurações de servidores *proxy*. A estratégia de verificação é baseada na criação de um modelo que expressa as configurações do *proxy* no formato de um autômato finito determinístico. Para a criação deste autômato, foram criados algoritmos para que seja feita a sua construção. Com a finalidade de verificar as configurações deste automato, foi definido um modo de entrada de propriedades à serem testadas para a validação do autômato.

O método proposto foi implementado originando a ferramenta `AclVerification`. ferramenta que é capaz de fazer a construção automaticamente do autômato a partir de um arquivo de configuração do *proxy* e também verificar propriedades diante deste autômato.

Foi realizado um estudo de caso aplicando a ferramenta a um arquivo de configuração de uma empresa de médio porte, este arquivo de configuração precisou ser adaptado pelo fato da ferramenta ainda não reconhecer os variados tipos de `acl`, que o Squid *proxy* possui. Aplicando o estudo de caso foi possível obter uma prova conceitual do método proposto, e também uma prova das funcionalidades da ferramenta.

Como trabalho futuro pretende-se estender os tipos de `acl` a serem verificadas pela ferramenta `AclVerification`. Pretende-se também alterar configurações para que o parser possa interpretar o arquivo de configuração do *proxy* por completo e ignorar informações desnecessárias para a verificação das regras. Espera-se também uma melhoria gráfica para que o usuário possa observar o automato graficamente, para que facilite ainda mais a correção do erro de configuração.

REFERÊNCIAS

[1] WESSELS, D. *Squid: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2004. (Definitive Guides). ISBN 9780596550530. Disponível em: <https://books.google.com.br/books?id=3xHKOYLKwlsC>.

[2] HUTH, M.; RYAN, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004. ISBN 9781139453059. Disponível em: <https://books.google.com.br/books?id=sVLOaObSBHkC>.

[3] BERARD, B. et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Berlin Heidelberg, 2001. ISBN 9783540415237. Disponível em: <https://books.google.com.br/books?id=j1WqQgAACAAJ>.

[4] JEFFREY, A.; SAMAK, T. Model checking firewall policy configurations. In: IEEE. *Policies for Distributed Systems and Networks, 2009. POLICY 2009. IEEE International Symposium on*. [S.l.], 2009. p. 60–67.

[5] YUAN, L. et al. Fireman: A toolkit for firewall modeling and analysis. In: IEEE. *2006 IEEE Symposium on Security and Privacy (S&P'06)*. [S.l.], 2006. p. 15–pp.

[6] ROUSSKOV, A.; SOLOVIEV, V. A performance study of the squid proxy on http/1.0. *World Wide Web*, Springer, v. 2, n. 1-2, p. 47–67, 1999.

[7] GARDENGHI, J. L. C.; BARDI, M. A. G. An authentication middleware for squid proxy-cache: a single sign-on approach. In: IEEE. *Computational Science and Its Applications (ICCSA), 2012 12th International Conference on*. [S.l.], 2012. p. 138–141.

[8] CÂMARA, E.; DEPUTADOS, C. dos. *Marco Civil da Internet - 2ª edição:*. Edições Câmara, 2015. (Legislação). ISBN 9788540203709. Disponível em: <https://books.google.com.br/books?id=PBlHCAAAQBAJ>.

[9] CLARKE, E.; GRUMBERG, O.; PELED, D. *Model Checking*. MIT Press, 1999. ISBN 9780262032704. Disponível em: <https://books.google.ca/books?id=Nmc4wEaLXFEC>.

[10] SAINI, K. *Squid Proxy Server 3.1: Beginner's Guide*. Packt Pub., 2011. ISBN 9781849513913. Disponível em: <https://books.google.com.br/books?id=HsbCb90HmlMC>.

[11] CHELLAS, B. *Modal Logic: An Introduction*. Cambridge University Press, 1980. ISBN 9780521295154. Disponível em: <https://books.google.com.br/books?id=YupiXWV5j6cC>.

[12] BLACKBURN, P.; BENTHEM, J. van; WOLTER, F. *Handbook of Modal Logic*. Elsevier Science, 2006. (Studies in Logic and Practical Reasoning). ISBN 9780080466668. Disponível em: <https://books.google.com.br/books?id=urINMvvsT5MC>.

[13] BAIER, C.; KATOEN, J. *Principles of Model Checking*. MIT Press, 2008. ISBN 9780262026499. Disponível em: <https://books.google.com.br/books?id=nDQiAQAAIAAJ>.

[14] FREITAS, C. de. *Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico*. Editora Feevale. ISBN 9788577171583. Disponível em: <https://books.google.com.br/books?id=zUDsAQAAQBAJ>.

[15] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML: guia do usuário*. CAMPUS - RJ, 2006. ISBN 9788535217841. Disponível em: <https://books.google.com.br/books?id=ddWqxcDKGF8C>.

[16] ASTAH. *Astah Professional*. 2016. Disponível em: <http://astah.net/editions/professional>.

[17] DEITEL, H. *Java: como programar*. PRENTICE HALL BRASIL, 2010. ISBN 9788576050193. Disponível em: <https://books.google.com.br/books?id=U5AyAgAACAAJ>.

[18] ORACLE. *Java SE Development Kit 8*. Acesso em: 07 de outubro de 2016. Disponível em: <http://www.oracle.com/technetwork/pt/java/javase/>.

[19] ORACLE. *Netbeans IDE 8.0.2*. Acesso em: 12 de julho de 2016. Disponível em: <https://netbeans.org/>.

[20] INITIATIVE, O. S. *The BSD 2-Clause License*. Acesso em: 15 de julho de 2016. Disponível em: <https://opensource.org/licenses/bsd-license.php>.

[21] PARR, T. *ANTLR 4.5*: Another tool for language recognition. Acesso em: 10 de dezembro de 2015. Disponível em: <http://www.antlr.org/>.

[22] GHAIDA, S. A. *IP/Subnet Calculator Lib*. 2014. Disponível em: <https://www.codeproject.com/tips/850531/ip-subnet-calculator-lib>.