CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
CELSO SUCKOW DA FONSECA – CEFET/RJ

Algoritmo de Autorização de Serviços baseado em Autômatos Finitos

Rodrigo Vidal Araujo

Prof. Orientador: Renato Mauro

Rio de Janeiro
Março de 2015

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
CELSO SUCKOW DA FONSECA – CEFET/RJ

Algoritmo de Autorização de Serviços baseado em Autômatos Finitos

Rodrigo Vidal Araujo

Projeto final apresentado em cumprimento às
normas do Departamento de Educação Superior
do CEFET/RJ, como parte dos requisitos para obtenção
do título de Tecnólogo em Sistemas para Internet

Prof. Orientador: Renato Mauro

Rio de Janeiro
Março de 2015

A663 Araujo, Rodrigo Vidal
Algoritmo de autorização de serviços baseado em autômatos finitos / Rodrigo Vidal Araujo – 2014
x, 33f.: il (algumas color.), grafs, ; enc.

Projeto Final (Tecnólogo) Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, 2014.
Bibliografia : f.3
Orientador : Renato Mauro

1. Engenharia de software. 2. Sistemas de parâmetros distribuídos. 3. Algoritmos computacionais. 4. NET framework (Tecnologia de rede de computador). I. Mauro, Renato (Orient.). II. Título.

CDD 005.1

DEDICATÓRIA

Dedico este trabalho a minha mãe, que nunca deixou de acreditar em mim, mesmo quando eu já não acreditava mais.

AGRADECIMENTOS

Ao Prof. Renato Mauro pela orientação deste trabalho, dedicação e ensinamentos transmitidos ao longo de projeto.

Aos professores do curso de Tecnologia em Sistemas para Internet do CEFET/RJ, minha maior gratidão por contribuírem para minha formação acadêmica.

Aos meus pais e familiares por apoiarem totalmente e acreditarem em minha capacidade.

RESUMO

Lidar com autorização de usuários e sistemas é comum em muitos sistemas de informação. Em cenários onde sistemas distribuídos conectados compõem uma solução única, este problema é ainda mais evidente porque cada um precisa checar as credenciais de acesso do usuário e do sistema que está realizando a requisição.

Neste trabalho é proposto um algoritmo de autorização baseado em autômatos finitos, onde se utiliza uma linguagem regular para definir quais recursos de um sistema distribuído podem ou não ser acessados.

Comparativamente, foi realizado testes junto à implementação padrão do .NET Framework, que se mostrou ineficiente para os casos que não requerem backtracking.

Palavras-chave: Autômatos, Autorização, Sistemas Distribuídos.

ABSTRACT

Dealing with authorization of users and systems is common in many information systems. In scenarios where distributed connected systems comprise a single solution, this problem is even more evident because each need to check the user's access credentials and the system that is performing the request.

This paper proposes an authorization algorithm based on finite automata, which uses a regular language to define which resources of a distributed system may or may not be accessed.

By comparison, testing was performed by the default implementation of the .NET Framework, which proved inefficient in cases that do not require backtracking.

Key-words: Automata, Authorization, Distributed Systems.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

REST – Representational State Transfer

NFA - Autômatos Finitos Não-Determinísticos

DFA - Autômatos Finitos Determinísticos

JSON – JavaScript Object Notation

Capítulo 1

Introdução

O presente projeto tem como tema “Algoritmo de Autorização de Serviços Baseados em Autômatos Finitos”, tema este bastante discutido e gerador de polêmicas atualmente.

Lidar com autorização de usuários e sistemas é comum em todos os sistemas de informação. Por autorização de usuários, entende-se, a capacidade de definir de forma unívoca, restrições de acesso a recursos disponibilizados. A grande maioria das modelagens realizam autorização, utilizando a estrutura relacional dos bancos de dados, enquanto neste trabalho apresentamos uma modelagem baseada em expressões regulares.

Cenários distribuídos, ou seja, em que vários computadores independentes entre si, interligados por uma rede de computadores, se apresentam ao usuário como um sistema único e coerente visando compartilhar a execução de tarefas, este problema é agravado. Pois cada um precisa verificar as credenciais de acesso do usuário e do sistema que está realizando a requisição, assim a lógica de autorização fica, em geral, espalhada nos diversos sistemas que compõem a solução.

A seguir buscar-se-á demonstrar a motivação e justificativa acerca do tema.

1.1 Motivação e Justificativa

Motivado pela crescente descentralização das unidades computacionais, onde cada uma passa a realizar operações cada vez mais especializadas, distribuídas geograficamente, foi decidido direcionar a análise às condições necessárias para que a comunicação entre esses nós, sejam permitidas caso respeitadas um conjunto de regras estabelecidas pelo negócio.

É importante notar que, para serem competitivas, é comum as empresas desenvolverem aplicações multicliente, em que a base de código e servidores são compartilhados por todos os seus clientes.

Vale ressaltar ainda, que a gestão de acesso aos recursos torna-se extremamente complexa caso, cada unidade de negócio, tenha que lidar com problemas de autorização, o que levaria a repetição de código e complexidade de manutenção em cada um desses serviços.

Sabendo-se que a autorização é uma preocupação ortogonal à especialidade de cada serviço, ela é tratada numa camada abaixo sem a necessidade da aplicação implementar

qualquer lógica de autorização. Precisando apenas, declarativamente, especificar a correspondência entre partes do sistema com as credenciais necessárias.

Neste aspecto o uso de expressões regulares, e sua implementação na forma de autômatos finitos, fornecem a sintaxe declarativa e a performance desejada para este tipo de aplicação.

1.2 Objetivos

A realização desse trabalho visa propor um algoritmo para avaliação de autorizações em uma arquitetura orientada a serviços. Para isso, será utilizado os conhecimentos adquiridos ao longo do curso de Tecnologia em Sistemas para Internet aliados à paradigmas, algoritmos e padrões para elaborar uma arquitetura de alta disponibilidade, escalável em um ambiente de computação em nuvem.

1.3 Metodologia e Trabalho Realizado

Primeiramente, será feita uma revisão bibliográfica mostrando os conceitos básicos de linguagens formais e teoria da computação.

Posteriormente, será apresentado como autômatos finitos podem compor uma solução simples para autorizar recursos, de modo similar a expressões regulares.

E por fim, um comparativo de performance em relação à biblioteca padrão de expressões regulares do .NET Framework.

1.4 Organização do Trabalho

O presente trabalho encontra-se dividido em 6 capítulos e trata-se de uma pesquisa descritiva, baseada em fontes primárias e secundárias.

Em um primeiro capítulo serão tratados temas acerca das considerações iniciais sobre a pesquisa. Para tanto, busca-se expor as motivações, justificativas, objetivos, a metodologia e a organização do presente trabalho.

O segundo capítulo trata de uma análise teórica do tema para uma melhor compreensão dos capítulos posteriores a este. Nele encontramos explicações acerca da Hierarquia de Chomsky, Máquinas de Estados e Autômatos (para uma análise mais

detalhada o tópico citado, encontra-se subdividido em Autômatos Finitos, Autômatos Finitos Determinísticos), Expressões Regulares e Construção de Thompson.

Em um terceiro capítulo, Implementação, é realizada a aplicabilidade da parte teórica anteriormente abordada.

No quarto capítulo é apresentado a avaliação experimental do projeto proposto.

Por fim, o último capítulo apresenta as conclusões finais e posteriores trabalhos que podem ser realizados para dar continuidade a esta pesquisa.

Capítulo 2

Fundamentação Teórica

Neste capítulo será introduzida um pouco da base teórica que guiou a implementação desde projeto.

Para tanto, cabe conceituar que um padrão é um conjunto de objetos com alguma propriedade reconhecível. Um exemplo de padrão é um conjunto de caracteres como o conjunto de identificadores permitidos na linguagem C.

Os dois problemas fundamentais associados com padrões são: a sua definição e o seu reconhecimento. Reconhecer padrões em programas é uma parte essencial para compilação e tradução de programas de uma linguagem - como a linguagem C- para outra - como a Linguagem de Máquina.

Existem muitos exemplos de uso de padrões em ciência da computação. Eles têm um papel fundamental no desenho de circuitos eletrônicos, usados para construir computadores e outros dispositivos digitais. São, ainda, utilizados em editores de texto para nos permitir buscar palavras específicas ou conjuntos de caracteres.

A maioria dos sistemas operacionais nos permitem usar padrões em seus comandos como o comando do UNIX *"ls *tex"*, por exemplo, que lista todos os arquivos cujos nomes terminem com a sequência *"tex".*

2.1 Hierarquia de Chomsky

Uma gramática formal pode ser classificada em um dos quatro tipos baseados em sua capacidade expressiva (CHOMSKY, 1957):

Tipo 0 – gramáticas recursivamente enumeráveis

Tipo 1 – gramáticas sensíveis ao contexto

Tipo 2 – gramáticas livres de contexto

Tipo 3 – gramáticas regulares

Cada conjunto é inclusivo no seu conjunto superior na hierarquia, com exceção das gramáticas livres de contexto que geram ϵ.

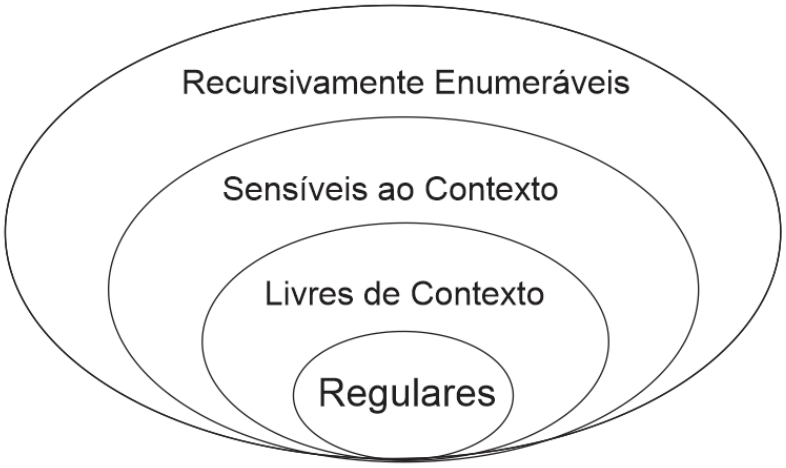


Figura 1 - Hierarquia de Chomsky

Neste trabalho, o será abordado o conjunto das linguagens regulares.

2.2 Máquinas de Estados e Autômatos

Máquinas de Estados e Autômatos são programas que procuram por padrões e, frequentemente, possuem uma estrutura especial. É possível identificar posições no código em que se conhece algo particular sobre o progresso do programa, afim de reconhecer uma instância de um padrão. Estas posições são chamadas de *estados*. O comportamento do programa pode ser visto como a *transição* de um estado para o outro conforme a *entrada* é lida.

Uma linguagem formal é um conjunto, potencialmente infinito, de palavras, que, por sua vez, são combinações de elementos de um conjunto finito de símbolos chamados de alfabeto.

Palavras são combinações entre os elementos de um alfabeto. Por exemplo: com um alfabeto $\sum = \{0,1\}$ é possível formar palavras que representam números binários. Uma palavra é representada pelos símbolos que a compõem. Logo, 01101 é uma palavra formada pelos símbolos do alfabeto descrito anteriormente. Sendo que a denotada por ϵ representa uma palavra vazia.

Linguagens são conjuntos de palavras sobre algum alfabeto. A mesma notação de conjuntos é utilizada por elas. Para melhor entendimento, vejamos: $A \ U \ B$ representa a

linguagem formada pela união de todas as palavras de A e B., assim como $\boldsymbol{w} \in \boldsymbol{A}$ é uma proposição que afirma que a palavra $\boldsymbol{w}$ está presente na linguagem A.

Existem duas formas principais para se definir linguagens formais: *Gramáticas gerativas* e *Autômatos.* A primeira, gera palavras da linguagem a partir de sistemas de reescrita e a última, por sua vez, reconhece palavras da linguagem.

A seguir, será tratado, de forma pormenorizada, fundamentos teóricos no que tange aos *Autômatos finitos, Autômatos finitos determinísticos* e *Autômatos finitos não-determinísticos*

2.2.1 Autômatos Finitos

Autômatos Finitos representam um modelo matemático de computação. Eles representam máquinas abstratas constituídas de um número finito de estados. Formalmente, autômatos finitos são uma 5-tuple (Q, $\sum$, δ, $\boldsymbol{q_0}$, F), onde:

- Q é um conjunto finito chamado de *estados.*
- $\sum$ é um conjunto finito chamado de *alfabeto*.
- $\delta : Q \times \sum \rightarrow$ é a *função de transição*.
- $\boldsymbol{q_0} \in Q$ é o *estado inicial*.
- $F \subseteq Q$ é o conjunto de estados de aceite.

Trata-se de reconhecedores, pois eles simplesmente dizem “sim” ou “não “sobre cada possível símbolo da entrada e possuem dois tipos: *Autômatos Finitos Determinísticos (DFA) e Autômatos Finitos Não-Determinísticos (NFA).*

Os *Autômatos Finitos Determinísticos (DFA), possui* exatamente, para cada estado e para cada símbolo do alfabeto da entrada, uma transição com este símbolo deixando este estado.

O segundo tipo, *Autômatos Finitos Não-Determinísticos (NFA)* não possuem restrições nos valores de suas transições. Um símbolo pode ser o valor de várias transições que saem do mesmo estado, e ϵ, a palavra vazia, é um valor possível.

Ambas as versões são capazes de reconhecer as mesmas linguagens, que de fato são exatamente as mesmas linguagens (chamadas de linguagens regulares), as quais as expressões regulares são capazes de descrever.

É possível representar um NFA ou DFA por um grafo de transições, onde os nós são estados e as arestas representam a função de transição. Existe uma aresta de valor *a*, do estado *s* para o estado *t*, se e somente se *t* é um dos próximos estados do estado *s* e entrada *a*.

2.2.2 Autômatos Finitos Determinísticos

Formalmente, autômatos finitos determinísticos são uma 5-tuple (Q, ∑, δ, $\boldsymbol{q_0}$, F), definidos em [1], onde:

- Q é um conjunto finito chamado de *estados*.
- ∑ é um conjunto finito chamado de *alfabeto*.
- δ : Q × ∑ → é a *função de transição*, que recebe como argumento o estado e um símbolo de entrada e retorna um estado. Na representação informal de um autômato como um grafo, δ é representado pelos arcos entre os estados e o valores destes arcos. Se q é um estado, e a é um símbolo de entrada então δ(q, a) é o estado p em que existe um arco com valor a de q para p.
- $\boldsymbol{q_0}$ ∈ Q é o *estado inicial*.
- F ⊆ Q é o conjunto de estados de aceite.

Especificar um DFA como uma 5-tupla com uma descrição detalhada da função de transição δ é tedioso e difícil de ler. Existem duas notações interessantes que podem facilitar a descrição de um autômato, que são: Diagramas de transição e Tabelas de transição.

Os Diagramas de transição são uma representação como um grafo. Para um DFA *A* é um grafo definido como:

- Para cada estado em Q haverá um nó.
- Para cada estado q em Q e cada símbolo de entrada a em ∑, faça δ(q, a) = p. Então ao digrama de transição tem um arco do nó q para o nó p, com valor a. Se existir múltiplos símbolos que causem transição de q para p, então o diagrama de transição pode possuir um arco, com esta lista de símbolos como valor.
- Existe uma flecha que entrada no estado inicial q0, chamado de Início. Esta flecha não tem início em nenhum nó.
- Nós correspondentes a estados de aceite (aqueles em F) são marcados por um círculo duplo. Estados que não estão em F possuem um círculo simples.

Exemplo: Para especificar um DFA que aceita todas as strings que possuem somente símbolos 0 e 1 e que possuem a sequência 01 em alguma posição pode ter o diagrama de transição descrito como:

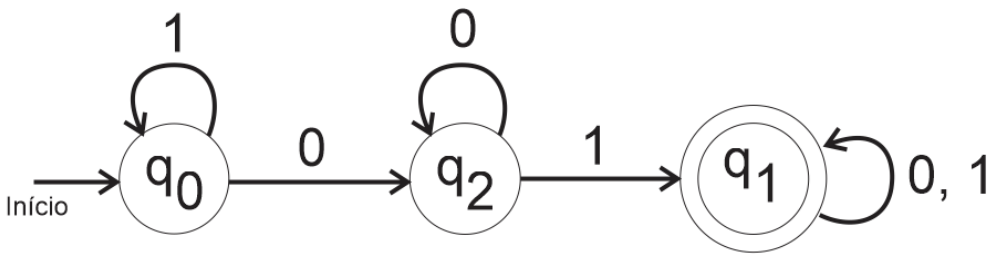


Figura 2 - Autômato Finito Determinístico

Uma descrição equivalente seria dizer que {x01y | x e y são qualquer conjunto de símbolos de 0's e 1's}.

Tabelas de transição são uma lista tabular de função δ - que implica dizer o conjunto de estados e o alfabeto de entrada. Trata-se de uma representação tabular onde a função δ recebe dois argumentos e retorna um valor. As linhas da tabela correspondem aos estados e as colunas corresponde às entradas. A entrada para a linha correspondente ao estado *q* e a coluna correspondente a entrada *a*, por exemplo, é o estado *δ(q,a)*.

A tabela de transição, correspondente a função δ do exemplo acima, é mostrado na Fig.

	0	1
→ q_0	q_2	q_0
* q_1	q_1	q_1
q_2	q_2	q_1

Figura 3 - Tabela de Transição de um DFA

O estado inicial é marcado com uma flecha, e os estados de aceite são marcados com um asterisco. Uma vez que é possível deduzir o conjunto de estados e símbolos de entrada olhando as linhas e colunas, é possível extrair da tabela de transição toda a informação necessária para especificar o autômato finito.

2.2.3 Autômatos Finitos Não-Determinísticos

Um autômato finito não-determinístico (NFA) tem a capacidade de estar em vários estados ao mesmo tempo. Um NFA aceita a mesma linguagem que é aceita por algum DFA, mas eles são geralmente mais sucintos e mais fáceis de projetar do que DFA's. Ainda é possível converter de um NFA para um DFA, mas este último pode possuir, exponencialmente, mais estados que um NFA. Felizmente, casos desse tipo são raros para a maior parte das aplicações.

Como uma DFA, um NFA possui um conjunto finito de estados, um conjunto finito de símbolos de entrada, um estado inicial, um conjunto de estados de aceite e, possui também, uma função de transição. A diferença entre um DFA e um NFA está no tipo da função δ. Para o NFA, δ é uma função que recebe um estado e um símbolo de entrada como argumentos, mas retorna um conjunto de zero, um, ou mais estados. A função do DFA, por sua vez, retorna exatamente um estado.

Exemplo de NFA seria reconhecer strings de 0's e 1's que terminem com 01:

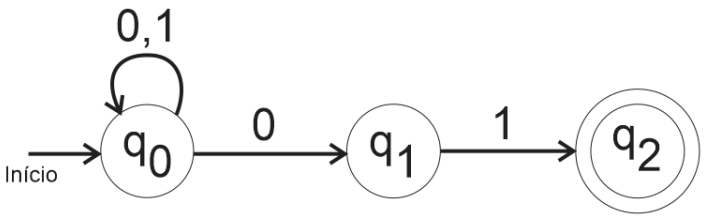


Figura 4 - Autômato Finito Não-Determinístico

Formalmente, autômatos finitos determinísticos são uma 5-tuple (Q, ∑, δ, $\boldsymbol{q_0}$, F), como definidos em [1], onde:

- Q é um conjunto finito chamado de *estados*.
- ∑ é um conjunto finito chamado de *alfabeto*.

- $\delta : Q \times (\sum\ U\ \{\epsilon\}) \rightarrow$ é a *função de transição*, que recebe como argumento o estado e um símbolo de entrada e retorna um subconjunto de Q. Repare que a diferença é o tipo do valor que δ retorna: um conjunto de estados no caso do NFA, e um único estado no caso do DFA.
- $\boldsymbol{q_0} \in Q$ é o *estado inicial*.
- $F \subseteq Q$ é o conjunto de estados de aceite.

Na forma de Tabela de Transição o autômato poderia ser representado da seguinte

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	∅	$\{q_2\}$
$*q_2$	∅	∅

Figura 5 - Tabela de Transição de um NFA

forma:

Repare que para o estado q0 a transição 0 leva a dois possíveis estados, q0 e q1.

2.3 Expressões Regulares

Expressões regulares, como definidas em [1], são sequências de caracteres que denotam linguagens regulares. Uma expressão regular de um alfabeto ∑ é definido como segue:

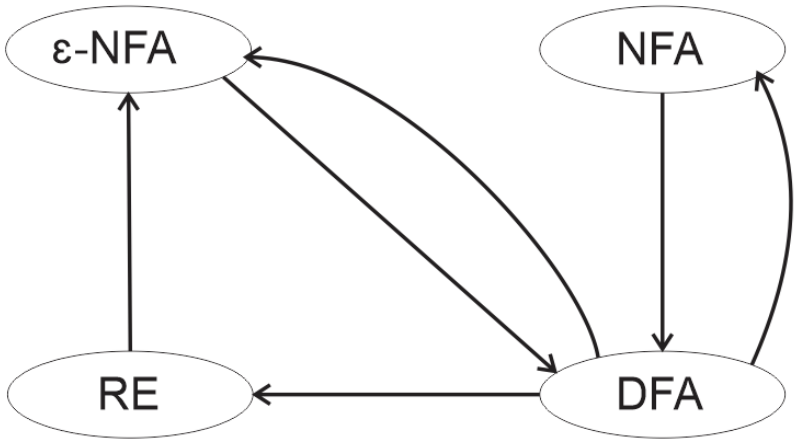
- ϵ, $\boldsymbol{\varphi}$, e a para cada a ∈ ∑ são expressões regulares denotadas pela linguagem regular, {ϵ}, ∅ (conjunto vazio), e {a} respectivamente.
- Se r1, r2 são expressões regulares denotadas pelas linguagens L1, L2, respectivamente, então (r1 | r2), (r1r2) e (r1*) são expressões regulares denotadas L1 U L2, L1L2 e L1*, que são chamadas de alternância, concatenação e *Kleene Closure*.

Todas as expressões regulares podem ser definidas com base nas regras acima.

Para cada símbolo em $\sum$ U $\{\epsilon\}$, uma ocorrência deste em uma expressão regular é chamada de literal. As Expressões regulares oferecem algo que autômatos, puramente, não conseguem: uma maneira declarativa de expressar quais são as strings que serão aceitas pelo autômato.

Embora a abordagem das expressões regulares seja fundamentalmente diferente da abordagem dos autômatos finitos, estas duas notações representam o mesmo conjunto de linguagens que são chamadas de linguagens regulares.

Todas as linguagens definidas por um autômato determinístico ou não-determinístico é também definida por uma expressão regular [3]. Além disso, toda linguagem definida por uma expressão regular também pode ser definida por esses autômatos.



2.4 Construção de Thompson

Para converter uma expressão regular em um NFA, pode-se utilizar o algoritmo de *McNaughton-Yamada-Thompson*. Este algoritmo funciona recursivamente dividindo uma expressão em suas partes constituintes, a partir do qual o NFA será construído usando um conjunto de modelos.

Figura 6 - Conversões entre as implementações de
Linguagens Regulares

Para ser mais preciso, cada subexpressão constrói o fragmento de um NFA e então a combinação desses fragmentos formam fragmentos maiores. É importante notar que estes fragmentos não são uma NFA completa. É preciso adicionar os componentes para formar um NFA completa.

Um fragmento consiste de um número de estados com transições entre estes e, adicionalmente, duas transições incompletas: uma apontando para o fragmento e outra saindo do fragmento.

Quando os fragmentos forem construídos para toda a expressão regular, a construção é completada conectando um estado de aceite à transição que sai do fragmento. A transição incompleta na entrada serve para identificar o estado inicial do NFA completo.

É possível o autômato obtido poder ser transformado em determinístico através do algoritmo conhecido como *Powerset Construction* e, então, minimizado para obter o autômato ótimo correspondente a uma expressão regular dada.

As regras são as seguintes:

Para a expressão ϵ constrói-se o NFA:

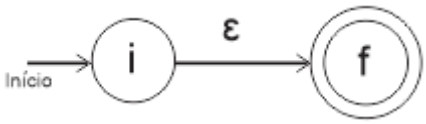


Figura 7 - NFA para expressão ϵ

Para a expressão a em ∑, constrói-se o NFA:

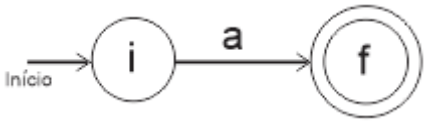


Figura 8 - NFA para expressão a

Para a expressão união (s|t) é convertido para:

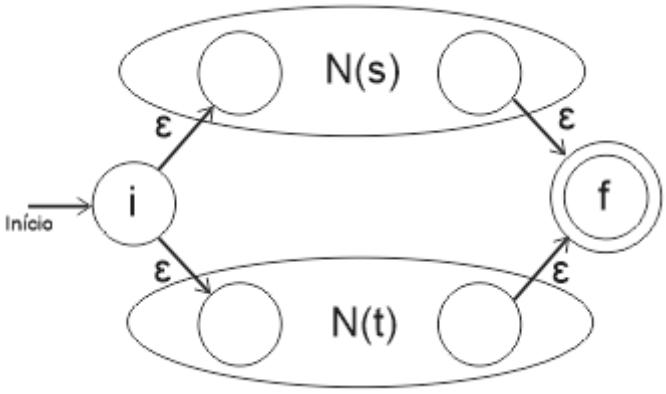


Figura 9 - NFA para expressão (s|t)

Onde N(s) representa a linguagem definida pela subexpressão s, e N(t) a linguagem definida pela subexpressão t, sendo assim, s|t define a linguagem que é a união entre as duas linguagens.

Para a expressão concatenação (st) constrói-se o NFA:

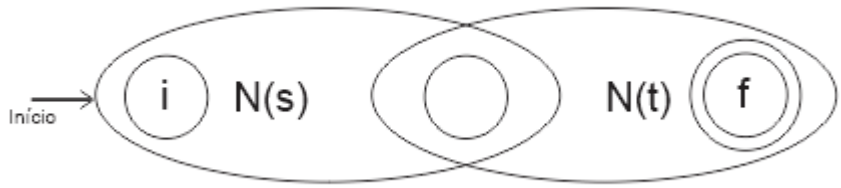


Figura 10 - NFA para expressão (st)

Para a expressão Kleene Closure s* constrói-se:

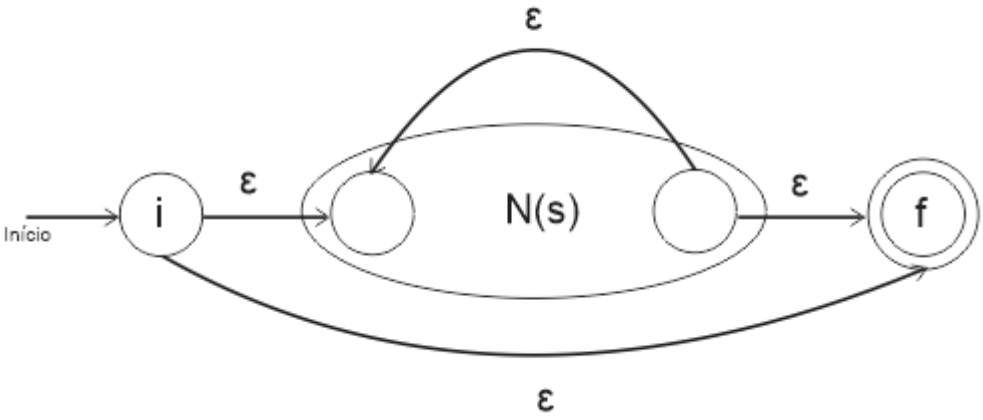


Figura 11 - NFA para expressão s*

2.5 Algoritmos de Busca para Expressões Regulares

Uma vez tendo o autômato finito não-determinístico é possível executá-lo usando uma string como entrada. A resposta será sim ou não. Para simular este autômato não-determinístico usando um computador convencional, é preciso encontrar uma maneira de "descobrir" qual transição deve ser seguida.

Uma maneira possível de simulação é tentar todos os caminhos possíveis, escolhendo um caminho inicial. Caso este não funcione, tenta-se um outro caminho. Esta abordagem é conhecida como *backtracking* e possui uma implementação simples e recursiva. Ela, no entanto, precisa ler a string de input muitas vezes antes de obter sucesso. Se a string não for aceita, ainda assim ela precisa tentar todos os possíveis caminhos de execução antes de desistir. No pior caso, pode haver um número de caminhos possíveis exponencial, levando a tempos de execução muito elevados.

Outra possível maneira de simular este autômato é executar vários caminhos simultaneamente. Nesta abordagem, a simulação permite que a máquina esteja em múltiplos estados ao mesmo tempo. Para processar cada caractere, todos os estados são avançados para aceitar este caractere. Esta abordagem lê cada caractere da string apenas uma vez. No pior caso, o NFA precisa estar em todos os estados a cada passo, mas esta quantidade independe do tamanho da string.

Dessa forma, strings grandes podem ser processadas em tempo linear. Esta é uma melhoria a se considerar, em comparação ao tempo exponencial requerido pela abordagem usando *backtracking*. Em uma NFA com *n* nós, pode haver no máximo *n* estados alcançáveis a cada etapa, mas pode haver $\mathbf{2^n}$ caminhos através da NFA. A eficiência vem do fato de nesse algoritmo ser mantido os estados alcançáveis, sem se importar com o caminho que foi usado para alcançá-los.

Autômatos finitos são mais fáceis de simular uma vez que se pode estar em apenas um estado por vez, e que há apenas uma transição possível para o caractere corrente.

Então, uma abordagem possível seria converter o NFA para uma DFA e então realizar a simulação. A conversão se dá pelo algoritmo conhecido com *Rabin-Scott Powerset Construction* (RABIN; SCOTT, 1959), que consiste em simular a execução paralela do autômato para todas as entradas possíveis. A principal vantagem é que, uma vez construído este autômato, é possível que se reconheça strings em tempo linear, mas, no entanto, a construção do DFA tem complexidades de tempo e espaço exponenciais e é bem mais complexa de ser implementada que as abordagens anteriores.

Capítulo 3

Implementação

No presente capítulo, será apresentada a implementação da biblioteca Automata (autômatos finitos em C#).

Neste projeto o objetivo é alcançar uma implementação funcional de um modelo de autorização baseado em autômatos finitos para permitir ou não requisições. A implementação é focada em ser simples, em detrimento de performance, por exemplo, não são feitas otimizações para redução do número de alocações de objetos.

A linguagem escolhida para implementar o projeto foi o C#, a qual trata-se de uma linguagem de máquina virtual - tal qual o Java - com resolução estática de tipos, o que oferece uma performance superior a linguagens com resolução dinâmica. Sua máquina virtual possui diversas otimizações.

O C# é uma linguagem que possui sintaxe similar ao C++ e permite uma implementação imperativa simples. Ela adere ao paradigma orientado a objetos, mas que também possui algumas construções do paradigma funcional que tornam a implementação mais expressiva.

O projeto não utiliza nenhuma biblioteca externa, apenas as bibliotecas padrões do *.NET Framework 4.5* e do C# 5.

3.1 O Problema

Em uma arquitetura orientada a serviços, vários pequenos serviços expostos via um modelo REST consomem uns aos outros. No entanto, é comum que serviços que não deveriam depender de outros acabem o fazendo, formando referências circulares e aumentando a complexidade da sua implementação. Logo, os serviços precisam ser autorizados, bem como os usuários.

Nesta arquitetura, para que o usuário consiga realizar uma operação 1 no serviço A, sabendo-se que tal operação depende de uma operação 2 no serviço B, é preciso que:

1. O usuário tenha permissão para acesso ao recurso 1 do serviço A.
2. O usuário tenha permissão para acesso ao recurso 2 do serviço B.
3. O serviço A tenha permissão para acesso ao recurso 2 do serviço B.

3.2 Funcionalidades Implementadas

Foi escolhido um conjunto de funcionalidades que pudesse dar ou retirar permissões de APIs. Para tal, utilizamos uma linguagem inspirada por expressões regulares e JSON.

Uma API é representada por dois valores:

- Um conjunto de recursos que possa ser representado por uma expressão regular.
- Um conjunto de operações que podem ser executadas nestes recursos. Vale dizer, que essas ações correspondem aos verbos REST e qualquer outra ação definida pelo usuário.

Para funcionalidades de alto nível foi implementado:

- Permitir acesso a uma API, especificando o recurso e suas ações (Allow).
- Negar acesso a uma API, também especificando o recurso e suas ações (Deny).

No que se refere ao reconhecimento da expressão regular foi implementado:

- Literais (e.g. a)
- Sequencias (e.g. ab)
- Alternâncias (e.g. ab|cd)
- Kleene* (e.g a*)

Estas funcionalidades são bem próximas as definidas por Thompson (THOMPSON, 1968) em seu artigo. Mas, como se pode notar, nem todas as funcionalidades foram implementadas, o que torna a linguagem ainda mais simples e apenas um subconjunto das linguagens regulares.

3.3 A Representação da API

Como dito anteriormente, para reconhecer padrões e autorizar os usuários, usamos a estrutura da própria requisição. Para dar permissão ao recurso *Orders*, que pode represetar uma API de pedidos, por exemplo, será usada a seguinte notação:

Effect = Allow

Resource = api/orders

Action = [GET, POST]

Esta política, como será chamada a estrutura formada por estas três partes, é capaz de informar ao sistema que um usuário ou sistema possuidor dessa política, pode realizar uma ação GET ou POST que são verbos HTTP, no recurso “api/orders”.

Compare com a política abaixo:

Effect = Deny

Resource = api/orders

Action = [GET, POST]

Como pode ser observado, esta política nega o acesso ao usuário ou sistema possuidor de realizar a ação GET ou POST no recurso “api/orders”.

Para autorizar se o agente da requisição pode ou não executar uma determinada operação, a ação que está sendo executada é verificada contra todas as políticas associadas a este agente. Então, pode-se dizer que as políticas que definem se o agente pode prosseguir ou não é um conjunto das triplas Effect, Resource, Action.

Para casos simples seria possível definir todas as operações que sua API expõe e, então, ao executar esta API, fazer uma simples comparação dessas strings. No entanto, é fácil verificar que esta opção não é interessante. A cada API adicionada seria necessário adicionar uma entrada nesse dicionário de operações, o que dificulta a manutenção do sistema e faz com que a lista cresça com cada novo recurso e subrecurso definido na API. Em um cenário onde vários serviços se comunicam através de numerosas e complexas API’s, essa solução não escalaria.

Para endereçar este problema é permitido o uso do asterisco (*) no valor do recurso. Este asterisco possui uma intenção diferente do asterisco encontrado em expressões regulares implementadas pela maioria dos frameworks disponíveis. Com o intuito de simplificar a linguagem utilizada pelo usuário para definir as regras, o asterisco tem como significado o Kleene Star da alternância de todos os símbolos presentes no alfabeto $\sum$.

Como exemplo, se o alfabeto $\sum$ for definido por $\{0,1\}$ então a expressão 0* equivaleria a expressão regular 0(0|1)*, que aceita qualquer string que inicia por 0 com qualquer número de caracteres.

3.4 Autômato para Avaliação de Políticas

Como descrito acima é possível ter políticas que fornecem permissão e outras que retiram. No entanto, o algoritmo precisa contemplar e unificar essas duas operações.

Para resolver esse problema pode-se pensar em dois autômatos: um para o cenário de Allow que será chamado de A e um para o estado de Deny que será chamado de D.

Como o autômato apenas informará se houve ou não um estado de aceite, a combinação destes dois autômatos é uma operação AND. Que segue:

- Caso A retorne positivo e D retorne positivo, a permissão é negada.
- Caso A retorne positivo e D retorne negativo, a permissão é dada.
- Caso A retorne negativo, a permissão é negada, independente do resultado de D.

Repare que para o caso de políticas que dão permissão (Allow), a omissão de um recurso é tratada como uma negação e para que se tenha acesso a um recurso, é necessário dar a permissão explicitamente, e não existir uma outra política que negue este acesso. Então:

- Por padrão, todos as requisições são negadas.
- Um Allow explícito sobrescreve o padrão.
- Um Deny explícito sobrescreve qualquer Allow.

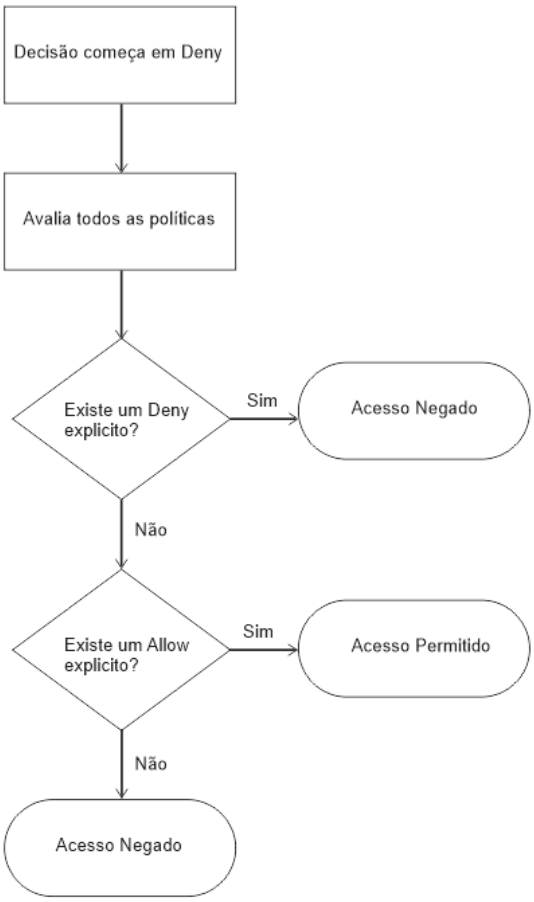


Figura 12 –Diagrama da regra de permissionamento

Através da figura abaixo, o assunto pode ficar melhor esclarecido:

3.5 Estrutura do Código

O código encontra-se dividido em três partes: um conversor para notação *postfix*, um *parser* para NFA e um *matcher*.

O conversor é responsável por receber uma string com a expressão regular no formato *infix*, e retornar uma nova string na notação *postfix*. A notação *infix*, prefixo e sufixo são formas de escrever expressões algébricas que se distinguem pela posição relativa aí de operadores e operandos. O operador está escrito diante de seus operandos na notação de prefixo entre seus operandos na notação *infix* e depois de seus operandos na notação *postfix*. Esta notação facilita a construção do autômato e é a mesma notação utilizada por Thompson em seu paper seminal.

O parser é responsável por obter a string em notação *postfix* e retornar o estado inicial do autômato a ser simulado.

O matcher, por fim, é responsável por saber simular a execução do autômato.

O uso desta biblioteca se dá pela compilação do autômato usando o construtor da classe Automata e a posterior avaliação de strings usando o método Match:

```
var nfa = new Automata(“abcd”);
var result = nfa.Match(s);
```

O resultado do método Match pode assumir apenas dois valores: verdadeiro ou falso. Este valor determina se a string, sendo avaliada, é aceita pelo padrão definido no construtor do autômato.

3.5.1 Compilando para um NFA

A notação *postfix* é uma notação matemática onde os operadores seguem os seus operandos. Seguindo o paper de Thompson o parser constrói um NFA a partir uma expressão regular em notação *postfix* com um ponto (.), adicionado explicitamente como um operador de concatenação. Por exemplo, para realizar a operação de adição tradicionalmente se escreve:

3 + 4

Em notação postfix seria:

3 4 +

No caso da implementação deste trabalho a expressão abaixo:

a(bb)+a

Seria traduzida pelo conversor em:

abb.+.a.

É importante notar que uma implementação POSIX (Portable Operating System Interface for uniX) precisaria utilizar o caractere ponto (.) como o metacaractere que denota qualquer caractere no lugar do operador de concatenação. No entanto, na gramática que será

utilizada aqui, este caractere não é permitido na entrada, logo não há qualquer problema em utilizá-lo para tal fim.

Outro ponto a se notar é que nesta notação o uso de parênteses não é necessário, pois o formato já leva em consideração a precedência destas operações. Mas ainda assim serão suportados.

O Parser, por sua vez, analisa a expressão *postfix*, e mantém uma pilha de fragmentos do autômato não-determinístico. Literais adicionam um novo fragmento na pilha, enquanto operadores removem fragmentos e então adicionam um novo.

No presente trabalho, o NFA será representando por um conjunto de coleções de estados encadeados. A classe que representa um estado é a State, mostrada abaixo:

```
public class State
{
    public StateType Type { get; set; }
    public char? C { get; set; }
    public List<State> Out { get; set; }
}
```

De acordo com a representação feita, cada estado representa um dos 3 fragmentos de NFA e, a seguir, fica dependente do valor da propriedade Type, que pode assumir os valores: Character, Divisão (Split) ou Aceite (Match):

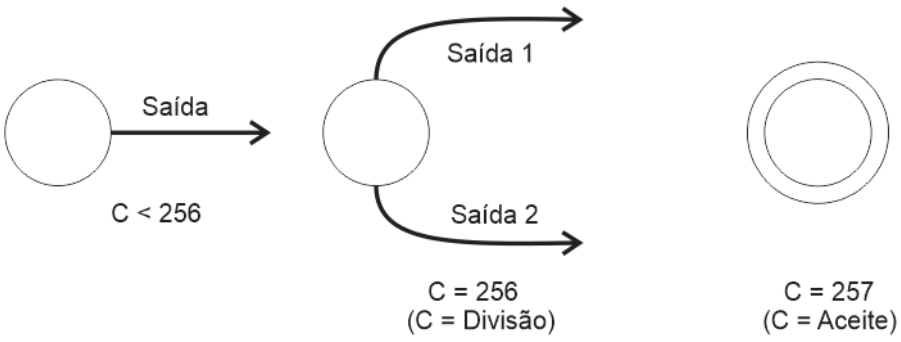


Figura 13 - Fragmentos do NFA

Após compilar os caracteres abb em abb.+.a., por exemplo, a pilha possuirá fragmentos para *a, b e b*. A compilação do ponto (.) que segue, irá remover dois fragmentos

do topo da pilha e irá colocar um novo fragmento para a concatenação bb. Cada fragmento é definido por um estado inicial e pelas transições de saída.

A classe que representa um fragmento do NFA é definido como:

```
public class Frag
{
    public State Start { get; private set; }
    public List<State> Out { get; private set; }
}
```

A propriedade Start representa o estado inicial do fragmento, enquanto que a propriedade Out é uma lista de estados que ainda não estão conectados a nenhum outro estado.

Dado estas primitivas e a pilha de fragmentos, o compilador é um laço de repetição sobre a expressão em notação postfix. No final há apenas um fragmento restante. Neste é adicionado um estado de Match e então o autômato está completo.

Perceba:

```
public static State Parse(string postfix)
{
    var stack = new Stack<Frag>();
    Frag e;
    Frag e1;
    Frag e2;
    State s;

    foreach (var p in postfix)
    {
        switch (p)
        {
            //cases
        }
    }
    e = stack.Pop();
    e.Patch(new State(StateType.Match));
    return e.Start;
}
```

Os casos tratados na cláusula switch são descritos abaixo:

Caracteres Literais:

```
default:
    s = new State(StateType.Character, c: p);
    stack.Push(new Frag(s, s));
    break;
```

```
case '.':
    e2 = stack.Pop();
    e1 = stack.Pop();
    e1.Patch(e2.Start);
    stack.Push(new Frag(e1.Start, e2.Out));
    break;
```

Concatenação:

Alternação:

```
case '|':
    e2 = stack.Pop();
    e1 = stack.Pop();
    s = new State(StateType.Split, o: e1.Start, o1: e2.Start);
    e1.Append(e2.Out);
    stack.Push(new Frag(s, e1.Out));
    break;
```

Zero ou Um:

```
case '?':
    e = stack.Pop();
    s = new State(StateType.Split, o: e.Start);
    e.Append(new List<State> { s });
    stack.Push(new Frag(s, e.Out));
    break;
```

Zero ou Mais:

```
case '*':
    e = stack.Pop();
    s = new State(StateType.Split, o: e.Start);
    e.Patch(s);
    stack.Push(new Frag(s, s));
    break;
```

Um ou mais:

```
case '+':
    e = stack.Pop();
    s = new State(StateType.Split, o: e.Start);
    e.Patch(s);
    stack.Push(new Frag(e.Start, s));
    break;
```

3.6 Simulação do NFA com Backtracking

Neste momento, é necessário testar se a expressão aceita a string de entrada. Para isso, a expressão regular é convertida em um NFA e então o NFA é simulado para a string de entrada.

Imagine que para simular um NFA com alternância, o autômato precisa testar para uma possibilidade e, caso esta não aceitasse a expressão, testar para a outra possibilidade. Esta característica é conhecida como *backtracking*. Por exemplo, para simular o NFA *abab* | *abbb* para a entrada *abbb*, o autômato teria a forma abaixo:

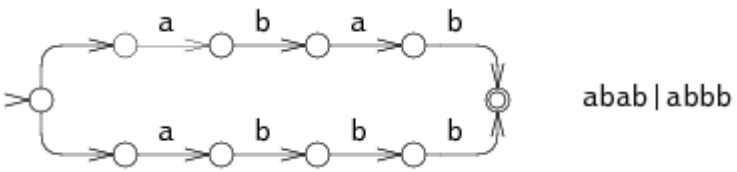


Figura 14 - Representação de Autômato com operação união [1]

O primeiro caso a se testar poderia ser *abab*, respeitando a precedência da esquerda para a direita. Para tanto, a string de entrada *abbb* iria até o primeiro estado após a transição *b*, e não atingiria um estado de aceite, logo falharia para este caso. Isto pode ser visto na imagem

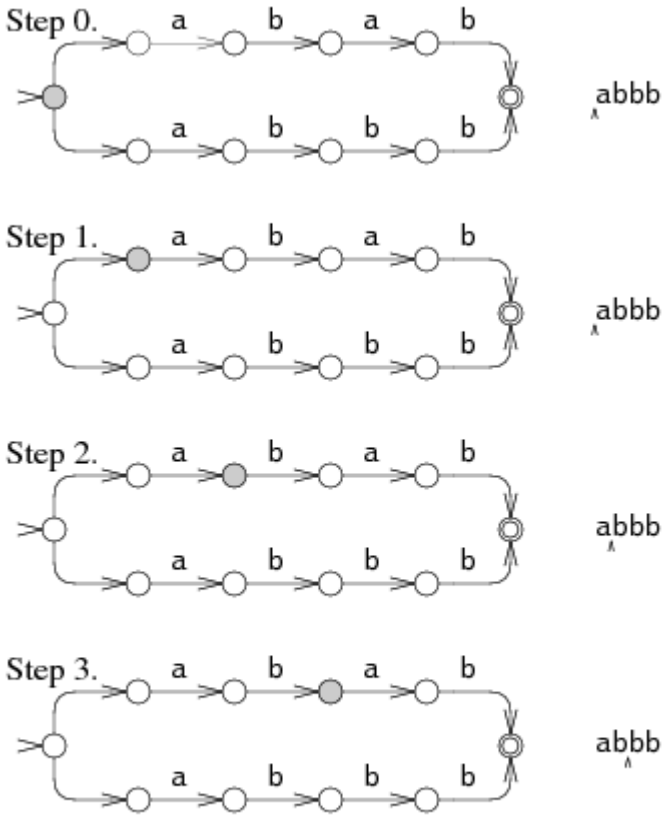


Figura 15 - Simulação do Autômato com Backtracking, primeiro caso [10]

abaixo:

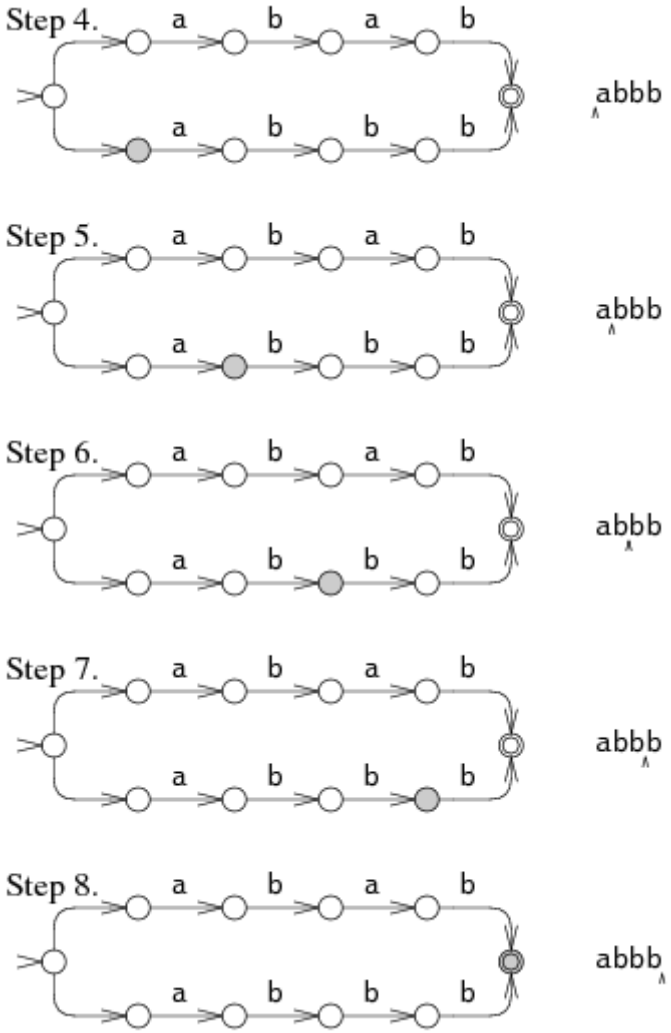


Figura 16 - Simulação do Autômato com Backtracking, segundo caso [10]

Esta abordagem tem uma implementação recursiva muito simples, mas pode ser que a entrada precise ser lida várias vezes até que o estado de aceite seja atingido. Se a entrada não for aceita, o autômato precisa testar a execução de todos os caminhos possíveis até desistir. No caso em tela, NFA tentou apenas dois caminhos diferentes, mas no pior caso, pode haver um número exponencial de caminhos de execução possíveis, levando a tempos de execução extremamente lentos.

3.7 Simulação do NFA em Paralelo

Uma maneira mais eficiente e complicada de “adivinhar” qual caminho seguir para simular o autômato é escolher as duas opções ao mesmo tempo. Nesta abordagem a simulação permite que o autômato esteja em vários estados de uma vez. Ao processar cada caractere da entrada, o autômato avança em todas as transições para este caractere.

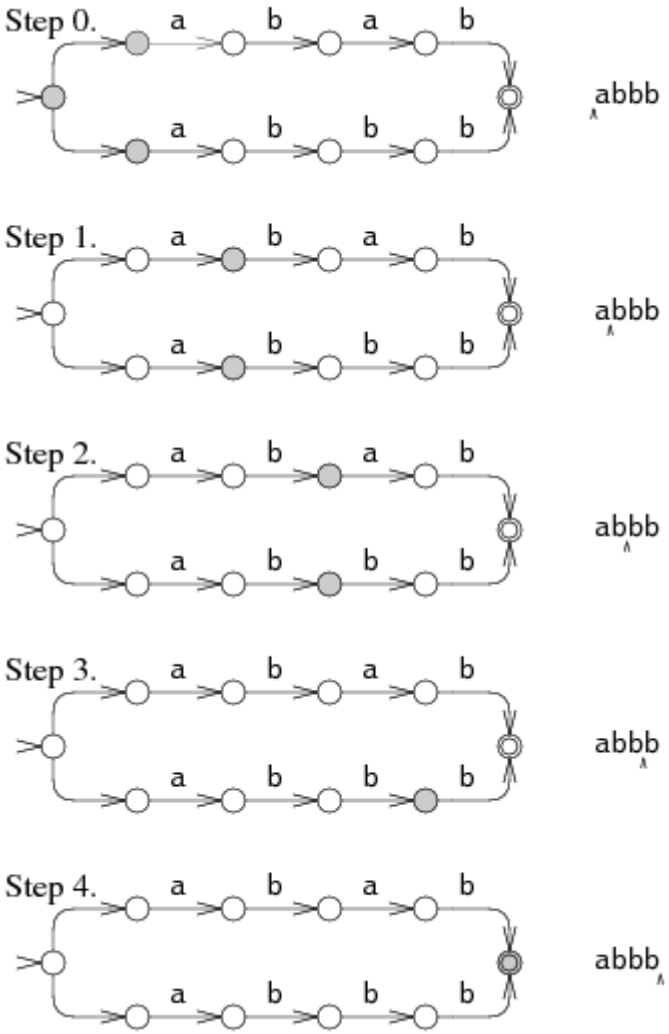


Figura 17 - Simulação do Autômato em paralelo [10]

Na imagem acima é possível ver que o autômato está em dois estados, simultaneamente, nas etapas 1, 2 e 3. Na etapa 4 a primeira opção é rejeitada e a segunda continua até chegar ao estado de aceite. Nesta implementação, cada caractere da entrada é lido apenas uma vez. No pior caso o NFA estaria em todos os estados ao mesmo tempo, mas isto resultaria em uma quantidade de trabalho que independe do tamanho da entrada, o que faz com que entradas grandes possam ser processadas em tempo linear. Esta implementação é bem mais adequada para os fins deste trabalho que a abordagem por backtracking que necessita de um tempo exponencial.

Thompson introduziu a simulação paralela em seu paper de 1968. Em sua formação, os estados do NFA eram representados por sequência de códigos de máquina e a lista de estados possíveis era um conjunto de instruções. Em sua essência, Thompson compilou a expressão regular em código de máquina.

Nos dias atuais, esta abordagem não é mais tão necessária. A seção a seguir descreve uma implementação em *C#* de uma maneira bem simples para executar esta simulação.

A simulação usa duas listas, sl é o conjunto atual de estados que o NFA se encontra e nl é o próximo conjunto de estados que o NFA estará após processar o caractere atual. O enlace inicializa sl apenas com o estado inicial e então inicia a simulação uma etapa por vez.

A simulação requer manter um conjunto de estados que são mantidos em um HashSet:

```
internal class StateList
{
    private HashSet<State> States { get; set; }

    public StateList()
    {
        States = new HashSet<State>();
    }
}
```

Se a lista de estados finais contém uma estado de aceite, então a string é aceita pelo autômato. O método que realiza essa verificação é o *IsMatch:*

```
public bool IsMatch()
{
    return States.Any(x => x.Type == StateType.Match);
}
```

Capítulo 4

Avaliação Experimental

Neste capítulo serão abordadas as principais diferenças entre a implementação proposta neste trabalho em comparação com a implementação padrão do .NET Framework. É importante notar que ambas as implementação foram escritas com a linguagem C# e sob a mesma versão do framework que é a versão 4.5. Ou seja, em termos de ambiente, as duas são equivalentes, o que torna mais justo a comparação, por se tratar apenas dos algoritmos.

Para demostrar os resultados foram feitos alguns testes utilizando a mesma expressão regular comparando seus tempos de execução em relação ao crescimento da string de entrada.

4.1 **Benchmarks**

Dado os testes abaixo, fica claro que a implementação do expressões regulares do .NET Framework utiliza backtracking, para todos os casos testados. É interessante notar que para vários casos de expressões regulares poderia haver uma otimização para não utilizar backtracking quando não fosse necessário. Se isso fosse feito em alguns caso a implementação seria polinomial ao invés de exponencial.

Nos benchmarks abaixo, o eixo das abcissas representa o tamanho da entrada, enquanto o eixo das ordenadas, representa o tempo de execução.

4.1.1 Benchmark 1: a^*b

O objetivo deste teste era mostrar o resultado mais básico de padrão com o uso de Kleene Star que é bem comum para definir padrões de aceite em recursos REST acessados por outros serviços.

O resultado mostrou que mesmo uma implementação bem simples do algoritmo conseguiu resultados mais rápidos que a implementação padrão do .NET. Como pode ser visto no gráfico a seguir:

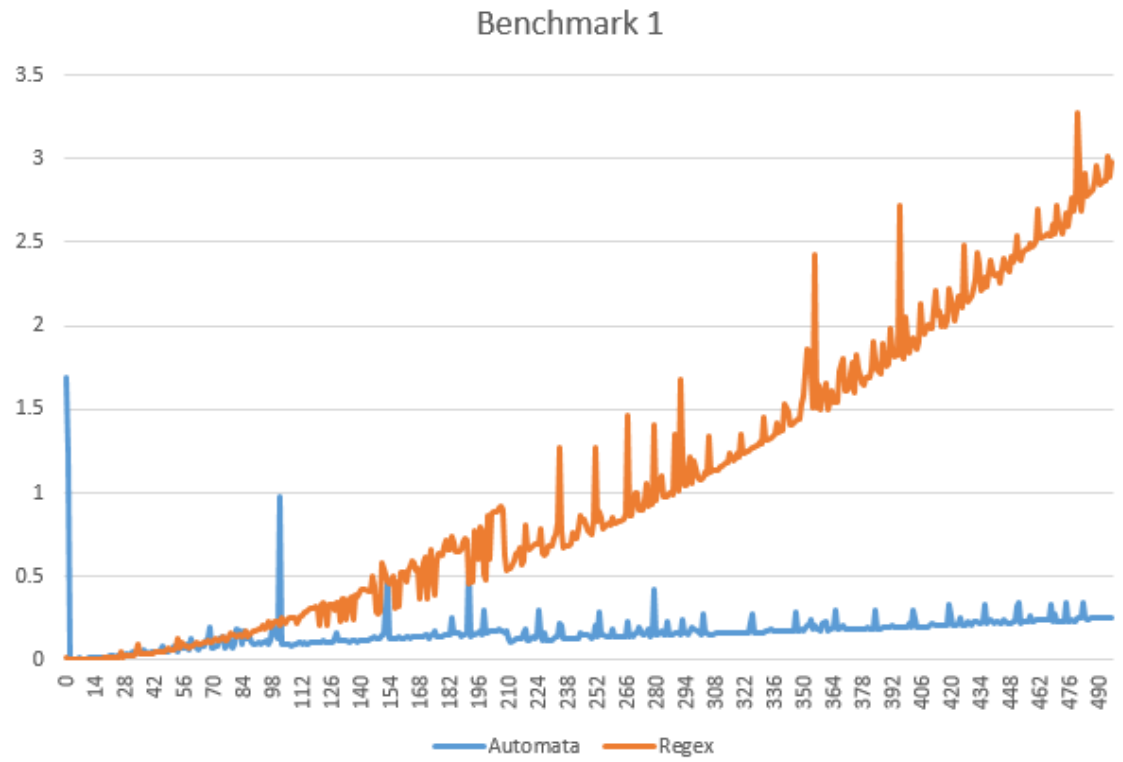


Figura 18 - Benchmark 1

4.1.2 Benchmark 2: $a^*a^*a^*a^*a^*b$

O objetivo deste teste era demonstrar um caso mais repetitivo, onde esta expressão força um backtracking para consume de várias repetições a*. Para este caso, onde a entrada é

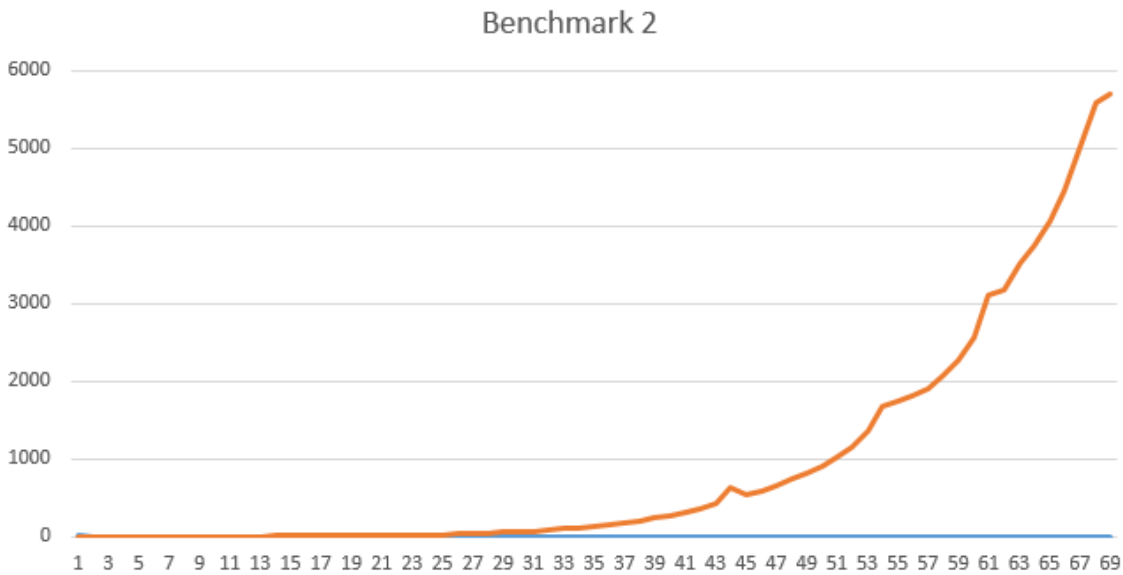


Figura 19 - Benchmark 2

uma concatenação de strings a, o resultado do .NET é ainda pior para o cenário em que a string não dá match no padrão acima. Perceba na imagem abaixo:

Enquanto isso, a implementação desde trabalho não requer muito tempo para mostrar que, a entrada não chega a um estado de aceite neste padrão.

4.1.3 Benchmark 3: $(a?)^n a^n$

Agora considere o expressão regular acima. Ela aceita a string a^n quando a? não aceita nenhum string deixando toda a string para ser aceita por a^n. Implementação baseadas em backtracking implementam zero ou um ? inicialmente testando para um e então zero. Existem n escolhas a se fazer de um total de 2^n possibilidades. Apenas a última possibilidade a ser verificada, que é escolher zero para todos os padrões (?), irá aceitar a string. A abordagem backtracking requer tempo O(2^n), então ela não escala muito além de n=25, como pode ser

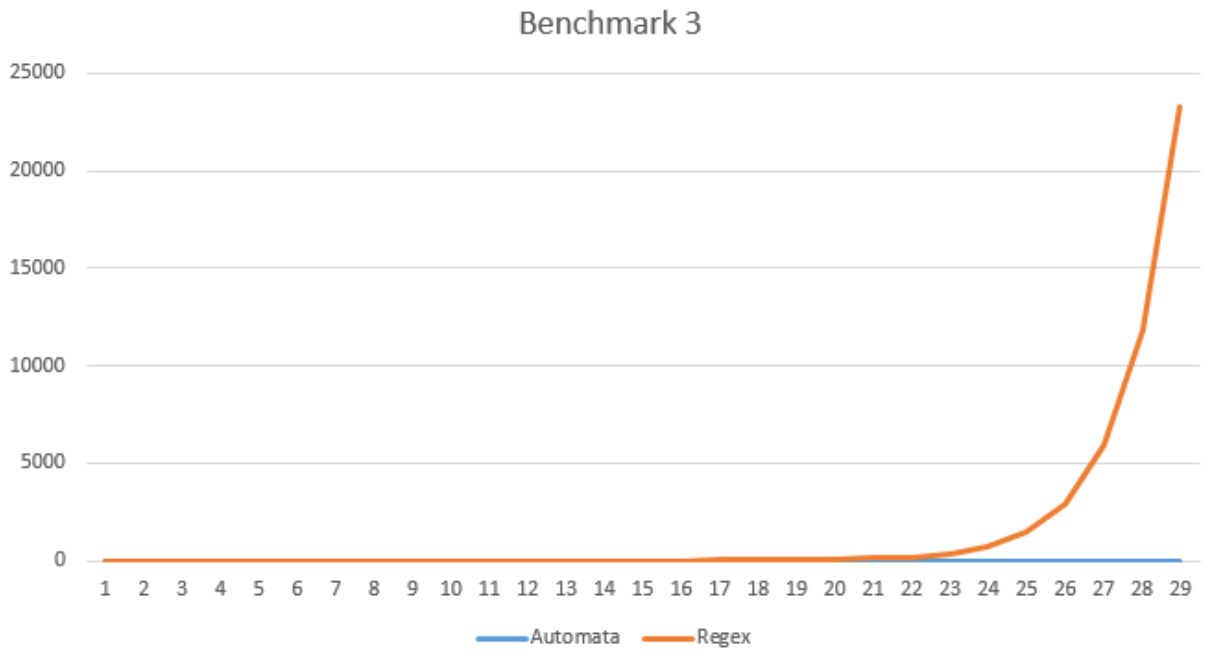


Figura 20 - Benchmark 3

visto no gráfico abaixo, para um match em a^n.

Capítulo 5

Conclusão

5.1 Contribuição

A principal contribuição neste trabalho foi a implementação de um motor de regras didático baseado em autômatos finitos para autorização de requisições feitas em uma arquitetura de serviços. Nela foi utilizada uma forma de representar um autômato finito não-determinístico que pode ser refeita em qualquer linguagem ou plataforma. A implementação foi feita em C# que é uma linguagem simples com sintaxe próxima ao C e Java, comum à maior parte dos estudantes de computação.

Os resultados alcançados foram satisfatórios, mostrando que é possível implementar um motor simples de autorização com uma sintaxe de expressões regulares. A implementação apesar de não ser a mais eficiente para casos triviais, mostrou-se muito útil para demonstrar didaticamente a teoria apresentada no capítulo 2.

Em COX2007 mostrou-se a incapacidade de implementações de expressões regulares em linguagens comuns como Python, Ruby e Java de reconhecê-las em tempo polinomial. O mesmo aconteceu para a implementação do .NET Framework, mesmo para expressões sem backreferences.

Estes resultados confirmam a impossibilidade de utilizar estas implementações para os cenários deste trabalho, ou em software que compartilham recursos de hardware entre vários usuários como sites, etc.

5.2 Trabalhos Futuros

A implementação apresentada no Capitulo 3 abordou os pontos mais básicos do reconhecimento de expressões regulares. Somente as estritamente necessárias para reconhecer cenários simples de autorização.

Nesta seção serão discutidas propostas de trabalhos futuros que podem dar continuidade à implementação apresentada de forma a torná-la apta ao uso em programas comerciais.

Implementar o algoritmo de uma maneira funcional: A grande vantagem de utilizar mecanismos que alteram estado é que este é facilmente implementado em qualquer linguagem imperativa. Como muito dos fragmentos de um autômato se repetem, uma proposta de estrutura de dados imutável que compartilhe estes fragmentos e ainda garanta uma implementação paralelizável.

Implementar a transformação para DFA: A implementação deste trabalho que não utiliza backreferences se mostra muito mais eficiente em tempo do que as implementações padrões da maior parte das linguagens. No entanto simular um NFA pode não ser o caminho mais eficiente. Transformar este autômato em um DFA para que futuras simulações utilizem deste mesmo DFA pode oferecer uma implementação mais eficiente.

Implementar posição do padrão que aceitou a entrada: A implementação atual, apenas informa se a entrada está ou não autorizada a realizar uma ação. Uma proposta de trabalho é identificar qual parte do padrão fez com que a entrada fosse aceita para que pudesse ser utilizada, por exemplo, para fins de log.

Referências Bibliográficas

[1] AHO, A; LAM M; SETHI, R.; ULLMAN, J; *Compilers Principles, Techniques, & Tools*, 2ª ed., California, Addison Wesley, 2007.

[2] COCHRAN, W; *Roll Your Own Regular Expression Engine*, Washington State University, Vancouver, 2006.

[3] HOPCROFT, J; MOTWANI, R; ULLMAN, J; *Introduction to Automata Theory, Languages, and Computation*, 2ª ed., Addison Wesley, 2001.

[4] HOLUB, J; *Simulation of NFA in Approximate String And Sequence Matching*, Czech Technical University, Czech Republic, 2013.

[5] GUANGMING, X; *Minimized Thompson NFA*, Western Kentucky University, United States, 2010.

[6] CALABRO, C; *NFA to DFA blowup*, 2005.

[7] THOMPSON, K; *Regular Expression Search Algorithm*, Bell Laboratories Inc., New Jersey, United States, 1969.

[8] MOGENSEN, T; *Introduction to Compiler Design*, 1ª ed., Springer, 2011.

[9] VEANUES, M; BJORNER, N; *Symbolic Automata: The Toolkit*, Microsoft Research, Redmond, WA, 2012.

[**10**] COX, R; *Regular Expression Matching, Can Be Simples and Fast*, http://swtch.com/~rsc/regexp/regexp1.html. Acessado em Julho de 2013.

[**11**] AHO, A; ULLMAN, J; *Foundation of Computer Science – C Edition;* W. H. Freeman, 1992