UNIVERSIDADE DO EXTREMO SUL CATARINENSE - UNESC

CURSO DE CIÊNCIA DA COMPUTAÇÃO

DIEGO POSSEBON FERNANDES

AF-GR WEB: UM APLICATIVO WEB, PARA MANIPULAÇÃO DE AUTÔMATOS FINITOS E GRAMÁTICAS REGULARES

CRICIÚMA

2014

DIEGO POSSEBON FERNANDES

AF-GR WEB: UM APLICATIVO WEB, PARA MANIPULAÇÃO DE AUTÔMATOS FINITOS E GRAMÁTICAS REGULARES

Trabalho de Conclusão de Curso, apresentado para obtenção de grau Bacharel no curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC.

Orientadora: Profª. MSc. Christine Vieira

CRICIÚMA

2014

DIEGO POSSEBON FERNANDES

AF-GR WEB: UM APLICATIVO WEB, PARA MANIPULAÇÃO DE AUTÔMATOS FINITOS E GRAMÁTICAS REGULARES

Trabalho de Conclusão de Curso aprovado pela Banca Examinadora para obtenção do Grau de bacharel, no Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC, com Linha de Pesquisa em Linguagens Formais.

Criciúma, 01 de julho de 2014.

BANCA EXAMINADORA

Christine Vieira

Profª. MSc. Christine Vieira - (UNESC) - Orientador

Prof. Esp. Luciano Antunes - (UNESC)

Silvana C. de Azevedo

Profª. MSc. Silvana Campos de Azevedo

AGRADECIMENTOS

Agradeço primeiramente a minha mãe, Carmen Lúcia Possebon Fernandes, pois sem ela nada disso seria possível. Com muita força e perseverança foi ela que me deu apoio, incentivou e nunca mediu esforços para que esse sonho se tornasse realidade. Por isso dedico a ela toda minha conquista.

A minha irmã, Denise Possebon Fernandes, pelas palavras de apoio e por sempre ter me incentivado na conclusão deste projeto.

Agradeço também a minha orientadora por ter me acompanhado e auxiliado em todas as etapas deste projeto. Ao professor Kristian Madeira, que sem obrigação alguma foi muito prestativo e contribuiu para uma parte fundamental do meu trabalho.

Aos meus familiares e amigos que de uma forma ou de outra sempre estiveram presentes e me incentivaram nessa trajetória.

E por fim ao meu pai, Jairo Pereira Fernandes, que ficou muito feliz quando entrei para a faculdade, mas que infelizmente não está mais entre nós para acompanhar a conclusão desta etapa. Fiz meu trabalho pensando em seus ensinamentos e espero que ele, onde estiver, possa sempre orgulhar-se de mim. E como ele sempre disse, tem mais Deus pra dar do que diabo pra tirar, um abraço a todos e até a páscoa.

RESUMO

Este trabalho de pesquisa resultou em uma aplicação WEB para manipulação de autômatos finitos e gramáticas regulares, tendo como base os algoritmos utilizados no software AFLAB. O software AFLAB apresentava alguns problemas que inviabilizaram sua utilização em sala de aula, como módulos não integrados, necessidade de instalação de mais de uma versão, não ter a opção de representar uma transição de um estado para o mesmo, não possuir setas nas arestas, complexidade na inserção de uma gramática regular e de um autômato finito na forma tabular. Foi então definido como objetivo desta pesquisa desenvolver uma aplicação WEB de manipulação de autômatos finitos e gramáticas regulares, baseado nos algoritmos utilizados no AFLAB, integrando os módulos e disponibilizando uma interface gráfica interativa para manipulação dos elementos. Na busca dos objetivos foi feito um estudo sobre os autômatos finitos, autômatos finitos com saída e as gramáticas regulares, além de uma análise nos algoritmos implementados no AFLAB. Foi desenvolvido então um aplicativo WEB onde o usuário pode desenhar autômatos finitos, testar sentenças, fazer a transformação de AFND para AFD, minimizar, gerar AF através de uma GR e gerar GR através de um AF. Ainda foi incluída a criação de máquinas de Mealy ou Moore, podendo fazer a simulação das maquinas com imagens como saída relacionada às transições (Mealy) ou aos estados (Moore). Por fim, temos uma aplicação WEB com um ambiente bastante interativo e de fácil utilização, que serve como um apoio no estudo de linguagens formais.

Palavras-chave: Autômatos Finitos, Gramáticas Regulares, Máquina de Mealy, Máquina de Moore, Linguagens Formais.

ABSTRACT

This research coursework resulted in a web application to handle finite automata and regular grammars, based on the algorithms used in AFLAB software. The AFLAB software had some problems and could not be used in the classes, like not integrated modules, need for installation of more than one version, not have the choice to represent a transition from a state to the same, not have arrows in the edges, complicacy in the insertion of a regular grammar and a finite automaton in tabular form. So, this research has as goal to develop a web application to manipulate finite automato and regular grammars based on the algorithym used in AFLAB software, integrating the modules and available an interactive grafic interface to manipulate the elements. In the search of the objectives, a study about the finite automata with exit and the regular grammars, beyound an analysis in the algorithym implemented in AFLAB was made. So, it was developed a web add where the user can draw finite automata, try to test sentences, do the transformation from finite automaton not deterministic to deterministic, minimize, create finite automaton through a regular grammar through a finite automaton. The creation of Mealy or Moore machines was included. They can do the machine simulation with images like outputs related to the transitions (Mealy) or the states (Moore). Lastly, we have a web application with a highly interactive ambience and easily use, that is like a support in the formal language study.

Key words: Finite Automata, Regular Grammars, Mealy Machines, Moore machines, Formal Language.

LISTA DE ILUSTRAÇÕES

Figura 1 - Diagrama de transição de um AFD. 14
Figura 2 - Diagrama de transição de um AFND 16
Figura 3 - AFD construído a partir do AFND. 17
Figura 4 - Diagrama de transição de uma MMe. 21
Figura 5 - Diagrama de transição de uma MMo. 22
Figura 6 - AF construído a partir da uma GR. 27
Figura 7 - Diagrama de transição do AFD para gerar a GR. 29
Figura 8 - Estrutura de Armazenamento de AF. 32
Figura 9 - Tipos declarados de Vetores. 32
Figura 10 - Função testa sentença. 33
Figura 11 - Função testa estado final. 34
Figura 12 - Função busca próximo estado. 34
Figura 13 - Código para geração do autômato finito a partir da gramática regular ...35
Figura 14 - Parte do código que monta a grade (Simulação Mealy). 36
Figura 15 - Parte do código que testa tipo de saída e sentença informada (Simulação Mealy). 36
Figura 16 - Parte do código que define a função de acordo com a saída (Simulação Mealy). 37
Figura 17 - Função que compara o símbolo de entrada com da transição (Mealy)...37
Figura 18 - Função de saída com imagem (Mealy). 38
Figura 19 - Função de saída com som (Mealy). 38
Figura 20 - Função de saída com texto (Moore). 39
Figura 21 - Declaração das classes de armazenamento do AFD. 40
Figura 22 - Tipos declarados. 40
Figura 23 - Identificação se a transição foi determinada. 41
Figura 24 - Definição do estado inicial no AFD. 42
Figura 25 - Inclusão do estado no AFD. 42
Figura 26 - Inclusão das transições dos estados no AFD. 43
Figura 27 - Eliminação de estados inacessíveis. 44
Figura 28 - Eliminação de estados mortos. 45
Figura 29 - Declaração Tclasse. 46

Figura 30 - Inserção estados finais e não finais. 46
Figura 31 - Identificação dos estados finais equivalentes. 47
Figura 32 - Identificação dos estados não finais equivalentes. 48
Figura 33 - Identificação dos estados equivalentes...49
Figura 34 - Gramática a partir de um AF. 50
Figura 35 - Funcionalidades do Simulador de Autômatos. 52
Figura 36 - Diagrama de caso de uso (Usuário)...54
Figura 37 - Diagrama de caso de uso (Sistema). 54
Figura 38 - Diagrama de caso de uso (Usuário) - Módulo autômato finito. 55
Figura 39 - Diagrama de caso de uso (Sistema) - Módulo autômato finito................55
Figura 40 - Diagrama de caso de uso (Usuário) - Módulo máquina de Mealy. 55
Figura 41 - Diagrama de caso de uso (Sistema) - Módulo máquina de Mealy. 55
Figura 42 - Diagrama de caso de uso (Usuário) - Módulo máquina de Moore. 56
Figura 43 - Diagrama de caso de uso (Sistema) - Módulo máquina de Moore. 56
Figura 44 - Classe Estado 57
Figura 45 - Classe Transição 57
Figura 46 - Função testa sentença. 59
Figura 47 - Transformação de AFND em AFD. 60
Figura 48 - Continuação transformação de AFND em AFD. 61
Figura 49 - Função eliminar estados inacessíveis. 63
Figura 50 - Método que define função total se necessário. 63
Figura 51 - Tabela com o relacionamento dos estados diferentes. 64
Figura 52 - Marcar estados trivialmente não equivalentes na tabela. 64
Figura 53 - Marcar estados não equivalentes na tabela...65
Figura 54 - Função que gera GR a partir de AF. 66
Figura 55 - Função que gera AF a partir de GR. 67
Figura 56 - Função que simula a MMe. 69
Figura 57 - Função que simula a MMo. 70
Figura 58 - Ambiente gráfico 71
Figura 59 - Tela opções de algoritmos. 72
Figura 60 - Tela reconhecedor de sentença...72
Figura 61 - Tela transformando AFND em AFD. 73
Figura 62 - Tela gramática regular - AF to GR. 74

Figura 63 - Tela gramática regular - GR to AF. 74
Figura 64 - Inserir imagem MMe. 75
Figura 65 - Inserir imagem MMo. 75
Figura 66 - Tela simulação autômatos com saída...76

LISTA DE TABELAS

Tabela 1 - Tabela de transição de um AFD. 14
Tabela 2 - Tabela de transição de um AFND. 16
Tabela 3 - Transformação em um AFD. 17
Tabela 4 - Tabela de relacionamento de estados diferentes. 19
Tabela 5 -Transições geradas a partir das produções.. 27
Tabela 6 - Transições do AFD para gerar a GR... 28
Tabela 7. Produções geradas a partir das transições.. 29

LISTA DE ABREVIATURAS E SIGLAS

AF	Autômato Finito
AFD	Autômato Finito Determinístico
AFND	Autômato Finito Não Determinístico
G	Gramática
GR	Gramática Regular
GLD	Gramática linear à direita
GLE	Gramática linear à esquerda
GLUD	Gramática linear unitária à direita
GLUE	Gramática linear unitária à esquerda
LF	Linguagens Formais
MMe	Máquina de Mealy
MMo	Máquina de Moore
WWW	World Wide Web

SUMÁRIO

1 INTRODUÇÃO 8
1.1 OBJETIVO GERAL 9
1.2 OBJETIVOS ESPECÍFICOS 9
1.3 JUSTIFICATIVA 10
1.4 ESTRUTURA DO TRABALHO 10
2 AUTÔMATOS FINITOS 12
2.1 AUTÔMATO FINITO DETERMINÍSTICO 13
2.2 AUTÔMATOS FINITOS NÃO DETERMINÍSTICOS 14
2.3 TRANSFORMAÇÃO DE AUTÔMATOS FINITOS NÃO DETERMINÍSTICOS EM DETERMINÍSTICOS 16
2.4 MINIMIZAÇÃO DE AUTÔMATOS FINITOS 17
2.5 AUTÔMATOS FINITOS COM SAÍDA 20
2.5.1 Máquina de Mealy 20
2.5.2 Máquina de Moore 22
3 GRAMÁTICAS REGULARES 24
3.1 GERAÇÃO DE UM AUTÔMATO FINITO A PARTIR DE UMA GRAMÁTICA REGULAR 26
3.3 GERAÇÃO DE UMA GRAMÁTICA REGULAR A PARTIR DE UM AUTÔMATO FINITO 27
4 AFLAB 30
4.1 ALGORITMOS UTILIZADOS NOS MÓDULOS DO AFLAB 31
5 TRABALHOS CORRELATOS 51
6 AF-GR WEB 53
6.1 METODOLOGIA 53
6.2 DESENVOLVIMENTO 56
6.2.1 Módulo Autômato Finito 58
6.2.1.1 Reconhecedor de sentenças 58
6.2.1.2 Transformação de AFND em AFD 59
6.2.1.3 Minimização de Autômatos Finitos 62
6.2.1.4 Gerar gramática regular a partir de um autômato finito. 65
6.2.2.4 Gerar autômato finito a partir de uma gramática regular. 66

6.2.2 Módulo Máquina de Mealy e Moore 68
6.2.2.1 Simular máquina de Mealy 68
6.2.2.1 Simular máquina de Moore 70
6.3 RESULTADOS OBTIDOS 71
7 CONCLUSÃO 77
REFERÊNCIAS 79
APÊNDICE A - ARTIGO 81

1 INTRODUÇÃO

A teoria das Linguagens Formais (LF) é parte da teoria da computação e, com tal, é imprescindível seu conhecimento por todos os profissionais e acadêmicos da área (ROSA, 2010). Isso acontece por que essa teoria é aplicada em análise léxica e análise sintática de diversas linguagens de programação utilizadas atualmente.

Um assunto abordado em LF é Autômato Finito (AF). O AF é um sistema de estados finito o qual constitui um modelo computacional do tipo sequencial muito comum em estudos de linguagens formais, compiladores, semântica formal e modelos para concorrência. Em LF este sistema é utilizado como reconhecedor de linguagens, que recebe como entrada uma cadeia de símbolos com o objetivo de mostrar se essa cadeia faz parte de determinada linguagem ou não. São classificados em Autômato Finito Determinístico (AFD) e Autômato Finito Não Determinístico (AFND). O AFD consiste que a partir de um estado corrente e do símbolo lido, o sistema pode assumir um único estado. Já o AFND pode assumir um conjunto de estados alternativos a partir de um estado e do símbolo de entrada. (MENEZES, 2005).

Uma extensão do AF são os autômatos finitos com saída denominados de máquina de Mealy e máquina de Moore. Com estas máquinas é possível gerar uma palavra de saída, ou seja, o autômato não fica limitado à aceita ou rejeita. Esses autômatos podem ser utilizados em aplicações como hipertexto, hipermídia e animação quadro-a-quadro, pois as saídas podem ser diversos tipos de dados como imagens, músicas, textos, entre outros (HOPCROFT; ULLMAN; MOTWANI, 2003).

Linguagem formal abrange também assuntos como gramáticas regulares, expressões regulares, minimização de autômatos finitos, transformação de autômatos finitos não determinísticos em determinísticos, geração de gramática regular a partir do autômato finito determinístico, entre outros. Diversos assuntos que necessitam de uma série de exercícios para melhor compreensão do aluno. Estes exercícios na maioria da vezes acabam sendo feitos a lápis e caneta, limitando os alunos a exemplos não muito complexos. Com objetivo de auxiliar o aluno nesse aprendizado, foi criado o grupo de pesquisa de linguagens formais do curso de ciência da computação da UNESC, que desenvolveu o AFLAB, software de simulação de autômatos finitos.

Segundo a professora da disciplina de LF alguns problemas foram detectados com a utilização do software em sala de aula. O AFLAB foi sendo incrementado com novos módulos de acordo com outros trabalhos desenvolvidos, onde alguns módulos acabaram não sendo integrados entre si, e para que se tenha acesso a todos os módulos desenvolvidos é necessária à instalação de três versões diferentes. Na elaboração do autômato na forma gráfica, este não tem a opção de representar uma transição de um estado para ele mesmo, ou seja, um laço. Além disso, não tem as setas nas arestas. Foi constatada também uma dificuldade na inserção de uma gramática regular ou de um autômato na forma tabular, pois a tela para entrada dos dados é muito complexa para o usuário. Estes problemas acabaram inviabilizando a utilização da ferramenta em sala de aula.

A partir do que foi observado, é proposto à criação de um aplicativo WEB, para manipulação de Autômatos Finitos e de Gramáticas Regulares, tendo como base os algoritmos utilizados nos módulos do AFLAB. O aplicativo irá integrar os módulos buscando as resoluções dos problemas apresentados, e terá como foco uma interface mais interativa e de fácil utilização, para que se torne uma opção de utilização mais viável dentro e fora da sala de aula.

1.1 OBJETIVO GERAL

Desenvolver um aplicativo WEB de manipulação de autômatos finitos e gramáticas regulares, baseado nos algoritmos utilizados no AFLAB.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos desse trabalho são:

a) entender os autômatos finitos;
b) compreender as Gramáticas Regulares;
c) integrar os módulos do AFLAB no novo aplicativo;
d) disponibilizar uma interface gráfica interativa para manipulação dos elementos no aplicativo.

1.3 JUSTIFICATIVA

A necessidade de um software que auxilie os alunos no processo de aprendizagem da disciplina de linguagens formais impulsionou o desenvolvimento de uma ferramenta denominada AFLAB para manipulação de autômatos finitos. Mas por problemas já apresentados no item 1, esta ferramenta acabou por não ser mais utilizada. Foi proposta então a criação de um aplicativo WEB baseado nos algoritmos implementados no software AFLAB, em que os alunos possam interagir na criação e manipulação de autômatos finitos e gramáticas regulares, resolvendo os problemas apresentados e tornando-o mais acessível.

A proposta de desenvolvimento do aplicativo WEB justifica-se por ser disponibilizada em uma plataforma que se destaca pelo seu grande potencial de interatividade, independência de localização e conectividade, acrescentando diversos benefícios ao mesmo. O acesso será mais fácil, onde qualquer pessoa com interesse no assunto poderá utilizá-lo, não limitando o uso somente a sala de aula. O aplicativo não necessitará de instalação e terá mais portabilidade, ou seja, pode ser utilizado em qualquer computador que possua internet e um browser, independente de sua configuração e sistema operacional.

Esses benefícios levam a uma fácil disponibilização e utilização do aplicativo, tornando mais viável seu uso em sala de aula e proporcionando aos professores da área de linguagens formais um ambiente bem interativo que pode estimular e auxiliar os alunos na compreensão do conteúdo, tornando a disciplina mais interessante no ponto de vista de quem estuda.

1.4 ESTRUTURA DO TRABALHO

O trabalho de pesquisa foi desenvolvido em seis capítulos. Primeiramente o capítulo 1 apresenta a introdução, os objetivos e a justificativa sobre o trabalho proposto.

No capítulo 2 é abordado os Autômatos Finitos, sua definição e conceito. Mostra que podem ser distinguido em duas classes, AFND e AFD, e como transformar um AFD em AFND. Explica o método de minimização de AF, deixando-o com o menor número de estados possíveis. E para terminar este capítulo temos os

autômatos finitos com saída, que são associadas às transições (Máquina Mealy) ou aos estados (Máquina de Moore).

As gramáticas regulares vêm logo a seguir no capítulo 3, explicando sua definição, finalidade e tipo de classificação através da hierarquia de Chomsky. Mostra como gerar uma gramática regular através de um autômato finito e um autômato finito através de uma gramática regular. Já no capítulo 4 temos o aplicativo AFLAB, onde foi apresentado um pequeno histórico de sua criação, finalidade, modo de desenvolvimento, estrutura, e por fim, os algoritmos utilizados em seus módulos. Os trabalhos correlatos ficaram no capítulo 5 que elenca alguns dos principais aplicativos desenvolvidos semelhantes ao AFLAB.

Por último temos o capítulo 6, que tem como finalidade apresentar o desenvolvimento do aplicativo proposto neste trabalho de pesquisa. Explica sua modelagem, funcionalidades e resultados obtidos. Na conclusão foram expostos as considerações finais e trabalhos futuros que podem ser desenvolvidos desta área.

2 AUTÔMATOS FINITOS

Nas décadas de 40 e 50 pesquisadores iniciaram estudos sobre os autômatos finitos, com o intuito de criar um modelo das funções do cérebro. Com isso, foi possível observar que esses simples tipos de máquinas poderiam ser um modelo útil para outras finalidades, como softwares que projetam e verificam comportamentos em circuitos digitais, analisadores léxicos para compiladores, programas que examinam grandes corpos de texto com o objetivo de localizar ocorrências de palavras, frases e outros padrões, e também para verificação de protocolos de comunicação ou protocolos para troca segura de informações. Devido ao relacionamento próximo dos autômatos finitos e as gramáticas formais, ou seja, um como reconhecedor e outro como conjunto de regras de uma linguagem, atualmente são a base de alguns importantes componentes de software, como no processo de análise sintática, análise semântica e geração de código dos compiladores atuais (HOPCROFT; ULLMAN; MOTWANI, 2003).

Um autômato finito consiste em um sistema de estados finitos, ou seja, um conjunto de estados finito e pré-definido onde cada estado tem apenas informações do passado, necessárias para definir as ações da próxima entrada. Podem ser definidos como reconhecedores de linguagens regulares ou expressões regulares, pois recebem uma sequência de caracteres como entrada, com o objetivo de conferir se a mesma pertence ou não à linguagem estabelecida (MENEZES, 2005).

Segundo Lewis (2004) autômato finito pode ser adotado como um modelo bem restrito de um computador real, que compartilha a característica de possuir uma unidade central de processamento de capacidade finita. Tem como entrada uma cadeia em forma de fita, onde não produz nenhuma saída, apenas indica se essa entrada foi ou não validada, ou seja, um reconhecedor de linguagem. O autor afirma também que o autômato finito não é totalmente isento de memória, apenas é limitada, fixada na fabricação e sem possibilidade de expansão posterior.

Menezes (2005) define um autômato finito como uma máquina composta, basicamente, de três partes:

a) fita: dispositivo de entrada que contem a informação a ser processada;

b) unidade de controle: reflete o estado corrente da máquina. Possui uma unidade de leitura (cabeça da fita) a qual acessa uma célula da fita de cada vez e movimenta-se exclusivamente para a direita;

c) programa ou função de transição: função que comanda as leituras e define o estado corrente da máquina.

Os autômatos finitos possuem duas classes com distinções fundamentais, em que em uma o controle pode ser “determinístico”, ou seja, não pode transitar para vários estados a partir do estado atual, e outra que é não determinístico, onde pode transitar para mais de um estado a partir do estado atual (HOPCROFT; ULLMAN; MOTWANI, 2003). Essas classes serão abordadas nos próximos dois subcapítulos abaixo.

2.1 AUTÔMATO FINITO DETERMINÍSTICO

Autômato finito determinístico (AFD) refere-se ao AF que para cada entrada possui somente um estado em que o autômato pode transitar a partir do estado em que se encontra. (HOPCROFT; ULLMAN; MOTWANI, 2003).

Conforme Menezes (2005), o autômato finito determinístico pode ser definido como uma quíntupla ordenada **M = (Σ, Q, δ, q0, F)**, na qual:

a) Σ é o alfabeto de símbolos de entrada;

b) Q é o conjunto de estados finitos;

c) δ é a função de transição, onde um estado P e um símbolo “a” resultam no estado q, ou seja, δ(p, a) = q;

d) q0 é o estado inicial, onde deve pertencer ao conjunto Q;

e) F é o conjunto de estados finais, onde devem pertencer ao conjunto Q.

Segundo Rosa (2010) podemos descrever o comportamento de um AFD M, com uma cadeia W em Σ, da seguinte maneira: M inicia em q0 e primeiramente busca o símbolo mais a esquerda de W. Com o símbolo encontrado e seu respectivo estado atual, M se desloca através da função de transição δ que mapeia os dois dados (estado/símbolo) para um novo estado. Com M no novo estado, é feita a leitura do segundo símbolo em W, e novamente uma transição δ ocorre. Este processo continua ocorrendo até que todos os símbolos em W sejam lidos. Após a leitura de todos os símbolos é verificado se o estado final é um elemento em F, onde

se pertencer a F a cadeia em questão é dada como aceita por M, e se não pertencer é dada como rejeitada.

Figura 1 - Diagrama de transição de um AFD.

Fonte: Do Autor.

Tabela 1 - Tabela de transição de um AFD.

δ	1	2
→ **q0**	q1	-
* **q1**	q0	q1

Fonte: Do Autor.

Os autômatos finitos podem ser representados de duas formas diferentes, uma seria utilizando um grafo ou diagrama de transição (figura 1), onde os nodos representam os estados, e os arcos são as transições entre esses estados. O estado inicial é apontado por uma seta, e os estados finais são marcados por circunferências concêntricas, ou seja, um círculo dentro no nodo. Outro modo seria através de uma tabela de transição de estados (tabela 1), contendo os estados nas linhas e os símbolos de entrada nas colunas. A tabela é preenchida pelas transições referentes a cada relação estado/símbolo de entrada. Para representar o estado inicial utilizamos uma seta (→), e para os estados finais um (*) (PRICE; TOSCANI, 2000).

2.2 AUTÔMATOS FINITOS NÃO DETERMINÍSTICOS

O não determinismo acrescenta um novo recurso ao autômato finito que consiste na capacidade de trocar de estados de forma parcialmente determinística partindo do estado atual e o símbolo de entrada, ou seja, permite a transição para

mais de um estado com uma única combinação estado/símbolo de entrada (LEWIS; PAPADIMITRIOU, 2004).

Segundo Menezes (2005), um autômato finito não determinístico (AFND) consiste em uma quíntupla ordenada **M = (Σ, Q, δ, q0, F)**, na qual:

a) Σ é o alfabeto de símbolos de entrada;

b) Q é o conjunto de estados finitos;

c) δ é uma função de transição parcial, onde um estado “p” e um símbolo “a” resultam em um conjunto de estados {q1, q2, ..., qn}, ou seja, δ(p, a) = {q1, q2, ..., qn};

d) q0 é o estado inicial, onde deve pertencer ao conjunto Q;

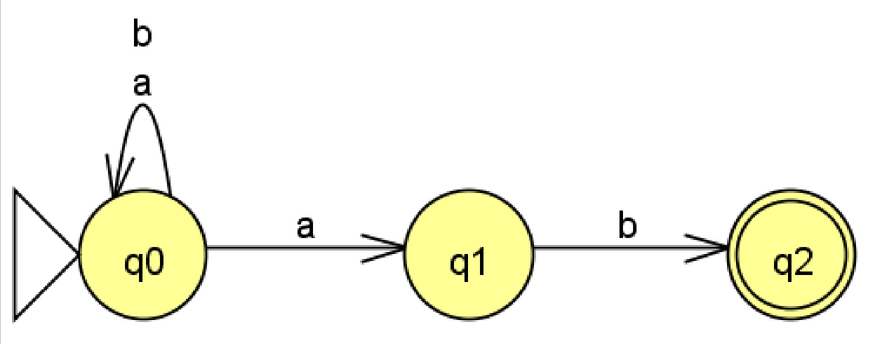
e) F é o conjunto de estados finais, onde devem pertencer ao conjunto Q.

Portanto, pode-se observar que um AFND tem a mesma definição que um AFD, exceto pela função de transição δ que os diferem (MENEZES, 2005).

De acordo com Rosa (2010) a diferença está em que o AFND contém um conjunto de estados possíveis, que através de determinada cadeia de símbolos de entrada pode levar a diversos caminhos, aonde alguns chegarão a um estado de aceitação, e outros a um estado de rejeição. Com isso, dizemos que uma cadeia de símbolos de entrada é dada como aceita pela linguagem se pelo menos um dos caminhos do AFND levar a um estado de aceitação.

A figura 2 representa um AFND na forma de diagrama, onde q0 é estado inicial, q2 é o estado final, e os símbolos de entrada pertencentes ao conjunto Σ são “a” e “b”. Pode ser observado que o não determinismo está no estado q0, pois partindo dele com o símbolo de entrada “a” duas opções de transição são possíveis, ou seja, pode ir para o estado q1 ou manter-se no mesmo estado. O mesmo pode ser observado na tabela de transição (tabela 2), a relação da linha do estado q0 com a coluna do símbolo de entrada “a” resultam em dois estados para transição, ou seja, uma função de transição δ(q0, a) = {q0, q1}.

Figura 2 - Diagrama de transição de um AFND



Fonte: Do Autor.

Tabela 2 - Tabela de transição de um AFND.

δ	a	b
→q0	q0, q1	q0
q1	-	q2
* q2	-	-

Fonte: Do Autor.

2.3 TRANSFORMAÇÃO DE AUTÔMATOS FINITOS NÃO DETERMINÍSTICOS EM DETERMINÍSTICOS

De acordo com Furtado (2004) a transformação de um AFND **M = (Σ, Q, δ, q0, F)** em um AFD **M' = (Σ', Q', δ', q0', F')**, deve seguir o seguinte teorema:

a) Q' = {ρ(Q)} - isto é, cada estado de M' será um subconjunto de estados de M;

b) qo' = [qo] - ou seja, qo' será o ρ(Q) composto apenas por q0;
   obs.: representaremos um estado q ∈ Q' por [q].

c) F' = {ρ(Q) | ρ(Q) ∩ F ≠ φ}

d) para cada ρ(K) ⊂ Q' definimos δ'(ρ(Q), a) = ρ'(Q), onde ρ'(Q) = {p | para algum q ∈ ρ(Q), δ(q, a) = p};

   ou seja, se ρ(Q) = [q1, q2, ..., qr] ∈ Q' e se

   δ(q1,a) = p1, p2, ..., pj

   δ(q2,a) = pj+1, pj+2, ..., pk

   :   :   :   :

   δ(qr, a) = pi, pi+1, ..., pn são as transições de M,

então ρ(Q) = [p1, ..., pj, pj+1, ..., pr, pi, ..., pn] será um estado de M', e M' conterá a transição: δ'(ρ(Q),a) = ρ'(Q).

Considerando o AFND na figura 2 a transformação ficaria conforme mostra a tabela 3:

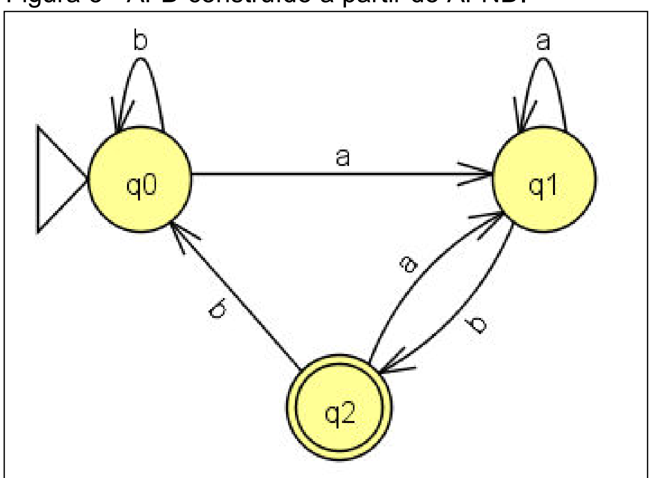
Tabela 3 - Transformação em um AFD.

δ	a	b
→q0	{q0, q1}	q0
{q0, q1}	{q0, q1}	{q0, q2}
***{q0, q2}**	{q0, q1}	q0

Fonte: do autor.

O diagrama que representa o AFND transformado em um AFD é ilustrado na figura 3. Para simplificar os estados {q0, q1} e {q0, q2} foram renomeados respectivamente para q1 e q2.

Figura 3 - AFD construído a partir do AFND.



Fonte: do autor.

2.4 MINIMIZAÇÃO DE AUTÔMATOS FINITOS

Rosa (2010) define o autômato finito determinístico mínimo ou autômato mínimo para determinada linguagem L, como um AFD M = (Σ, Q, δ, q0, F) tal que M aceite L, e que para qualquer outro AFD M' = (Σ', Q', δ', q0', F') tal que M também aceite L, ocorra que Q ≥ Q'.

O termo mínimo é empregado para designar um autômato finito que tenha o número mínimo possível de estados, ou seja, para um AF ser dado como mínimo

não pode possuir estados inacessíveis, inúteis, e equivalentes. Os inacessíveis são estados pertencentes ao conjunto de estados que a partir do estado inicial não há qualquer caminho para chegar até ele. Já os inúteis são estados que também pertencem ao conjunto de estados, onde não são um estado final e a partir dele nenhum estado final pode ser alcançado. E por fim, os equivalentes são um conjunto de estados que pertencem a uma mesma classe de equivalência, ou seja, um conjunto de estados q1, q2, ..., qi está em uma mesma classe de equivalência se δ(q1, a), δ(q2, a), ..., δ(qi, a), para cada "a" ∈ ∑, resultarem respectivamente nos estados qi, qi+1, ... qn (FURTADO, 2004).

Conforme Menezes (2005) para minimizar um autômato finito deve ser observado se o mesmo atende aos seguintes pré-requisitos:

a) é determinístico;
b) todos os estado devem ser alcançáveis a partir do estado incial, ou seja, não pode ter estados inacessíveis;
c) a função de transição deve ser total, ou seja, partindo de qualquer estado, possui transições para todos os símbolos do alfabeto.

O processo de minimização deve seguir o seguinte algoritmo, onde tem como entrada um AFD M = (Σ, Q, δ, q0, F), e retornará um AFD M' = (Σ', Q', δ', q0', F') | M' ≡ M (FURTADO, 2004):

a) eliminar os estados inacessíveis;
b) eliminar os estados inúteis;
c) construir todas as possíveis classes de equivalência de M.
d) construir M', como segue:
   - Q': é o conjunto de classe de equivalência obtida,
   - q0': é a classe de equivalência que contem q0,
   - F': é o conjunto das classes de equivalência que contenham pelo menos um elemento ∈ F, ou seja, { [q] | ∃ p ∈ F em [q], onde [q] é uma classe de equivalência },
   - δ' - δ'([p], a) = [q] ⇔ δ(p1, a) = q1 é uma transição de M ∧ p1 e q1 são elementos de [p] e [q] respectivamente.

Para eliminação dos estados inacessíveis deve se seguir o seguinte método:

a) primeiramente é marcado o estado inicial de M;

b) para cada estado q_i marcado, marcar todos os estados q_n que podem ser alcançados por uma transição a partir de q_i;
c) todos os estados não marcados devem ser eliminados.

Para eliminação dos estados inúteis deve se seguir o seguinte método:

a) primeiramente marcar os estados finais;
b) marcar todos os estados q_n que alcançam um estado marcado q_i por uma transição com qualquer símbolo de entrada;
c) eliminar os estados não marcados, deixando indefinidas as transições que levam a um estado eliminado.

Para eliminação dos estados equivalentes deve se aplicar os seguintes passos (ROSA, 2010):

a) tabela: construir uma tabela (tabela 5) que deve ser feito o relacionamento dos estados diferentes, onde cada par de estados ocorre somente uma vez;

Tabela 4 - Tabela de relacionamento de estados diferentes.

q1				
q2				
...				
qn				
	q0	**q1**	**...**	**qn**

Fonte: do autor.

b) marcar na tabela todos os estados obviamente não equivalentes, estados finais e não finais;
c) marcação dos estados equivalentes: Para cada par {q, q'} e para cada símbolo a ∈ ∑, suponha que **δ(q, a) = p** e **δ(q', a) = p'** e:
   - se *p = p'* , então q é equivalente a *q'* para *a* e não deve ser marcado,
   - se *p ≠ p'* e o par *{p, p'}* não está marcado, então *{q, q'}* é incluído em uma lista para análise posterior,
   - se *p ≠ p'* e o par *{p, p'}* está marcado:
     - *{q, q'}* não é equivalente e deve ser marcado,
     - se *{q, q'}* encabeça uma lista de pares, então marcar todos os pares da lista;
d) unificação dos estados equivalentes: os pares de estados não marcados na tabela são equivalentes e são unificados como segue:

- a equivalência é transitiva,
- se os pares de estados forem não finais equivalentes à unificação é feita em um único estado não final,
- se os pares de estados forem finais equivalentes à unificação é feita em um único estado final,
- se algum dos estados equivalentes for inicial, então o estado unificado é inicial também;

2.5 AUTÔMATOS FINITOS COM SAÍDA

Os autômatos finitos são limitados em relação à utilização em aplicações, isso devido a sua saída ser basicamente uma lógica binária aceira/rejeita. Com isso, é possível estender os autômatos para que possam gerar uma cadeia de saída, sem alterar suas propriedades. Para que isso seja possível são feitas modificações sobre os autômatos finitos e suas saídas são associadas às transições (Máquina Mealy) ou aos estados (Máquina de Moore). Com esse incremento essas máquinas tiveram destaque na utilização em aplicativos focados para World Wide Web (WWW), como aplicativos hipertexto, hipermídia e animações quadro-a-quadro (MENEZES, 2005).

De acordo com Menezes (2005) a saída nas duas máquinas não pode ser lida, isto é, não serve como memória auxiliar. A saída é como segue:

a) definida em um alfabeto chamado de alfabeto de símbolos de saída, onde pode ser igual ao alfabeto de símbolos de entrada;
b) armazenada em uma fita de saída, independente da fita de entrada;
c) a cada símbolo a cabeça da fita de saída move uma célula pra a direita;
d) o resultado do autômato é se a sentença foi aceita ou rejeitada e a informação contida na fita de saída.

2.5.1 Máquina de Mealy

A máquina de Mealy nada mais é que uma modificação no autômato finito com o objetivo de gerar uma saída, que pode ser vazia também, para cada transição (MENEZES, 2005).

Conforme Rosa (2010) a máquina de Mealy (MMe) consiste em um AFD onde suas saídas são ligadas as transições. É representado por uma sêxtupla **M = (Q , Σ, δ, q0, F, ∆)**, onde:

a) Q é o conjunto finito de estados;

b) Σ é o alfabeto de símbolos de entrada;

c) δ é uma função de transição de estado, definida por **Q x Σ → Q x** ∆*****;

d) q0 é o estado inicial, onde deve pertencer ao conjunto Q;

e) F é o conjunto de estados de aceitação, onde devem pertencer ao conjunto Q;

f) ∆ é o alfabeto de símbolos de saída.

Observa-se que os componentes da MMe são basicamente um autômato finito determinístico, diferem apenas na função de transição e no acréscimo do alfabeto de símbolo de saída. Portanto, a representação gráfica pode ser feita também por um diagrama de transição, acrescentando apenas a saída na etiqueta da transição em que ela está associada. Quando a saída for vazio não é necessário associar a transição (MENEZES, 2005). A figura 4 apresenta a representação gráfica de uma MMe.

Figura 4 - Diagrama de transição de uma MMe.

Fonte: do autor.

Segundo Menezes (2005) para uma entrada w pertencente Σ, a máquina vai processar a função de transição para cada símbolo em w, seguindo da esquerda para direita, até ocorrer uma parada. Se tiver a palavra vazia como saída da função de transição nada ocorre, ou seja, nenhuma gravação é feita na fita e consequentemente a cabeça da fita de saída não se move. Se em todas as transições tiverem como saída uma palavra vazia, a MMe processa exatamente do jeito de um autômato finito.

2.5.2 Máquina de Moore

A máquina de Moore possui uma segunda função associado a cada estado da máquina com o intuito de gerar uma saída, que pode ser vazia ou não (MENEZES, 2005).

De acordo com Rosa (2010) a máquina de Moore (MMo) consiste basicamente em um AF com saídas ligadas aos estados, e é definida por uma sétupla **M = (Q , Σ, δ, q0, F, ∆, δs)**, onde:

a) Q é o conjunto finito de estados;

b) Σ é o alfabeto de símbolos de entrada;

c) δ é uma função de transição parcial, definida por **Q x Σ** ⟶ **Q**;

d) q0 é o estado inicial, onde deve pertencer ao conjunto Q;

e) F é o conjunto de estados de aceitação, onde devem pertencer ao conjunto Q;

f) ∆ é o alfabeto de símbolos de saída;

g) δs é a função total de saída, ou seja, **Q** ⟶ ∆*.

Portanto, seu funcionamento é praticamente igual à de um AFD. A diferença está em que o AF apenas retorna uma saída binária aceita/rejeita, e na MMo estará presente uma palavra de saída, onde é inicializada por δs(i), e quando ocorre uma transição para um estado, por exemplo q0, δs(q0) é inserido a direita da fita de saída, ou seja, vai preenchendo a fita de saída da esquerda para direita com o símbolo de saída referente ao estado alcançado (VIEIRA, 2006). A figura 5 apresenta a representação gráfica de uma MMo.

Figura 5 - Diagrama de transição de uma MMo.

Fonte: Do autor.

Conforme Menezes (2005), para uma entrada w pertencente a Σ, o funcionamento de uma MMo consiste em aplicar a função de transição para cada símbolo de w, da esquerda para direita, efetuando também a função de saída para cada estado transitado. Se tiver a palavra vazia como saída da função de transição nada ocorre, ou seja, nenhuma gravação é feita na fita e consequentemente a cabeça da fita de saída não se move. Se em todas as transições tiverem como saída uma palavra vazia, a MMo se comportará da mesma forma que um autômato finito.

3 GRAMÁTICAS REGULARES

Essencialmente, uma gramática é um conjunto de regras finitas, que quando aplicadas continuamente, geram palavras. O conjunto dessas palavras geradas pela gramática constitui uma linguagem (MENEZES, 2005).

Menezes (2005) define a gramática como uma quádrupla ordenada **G = (V, T, P, S)**, onde:

a) V é um conjunto finito de símbolos não terminais;
b) T é um conjunto finito de símbolos terminais;
c) P é uma relação finita, ou seja, conjunto finito de pares denominado produções **(α, β)**, onde cada par é uma regra da produção.
d) S é o símbolo inicial pertencente a V.

De acordo com Furtado (2004) a gramática tem como finalidade definir uma estrutura sobre um alfabeto de maneira a possibilitar que apenas determinadas combinações sejam alcançadas, essas combinações são consideradas sentenças, definindo assim a linguagem que ela representa. Informalmente pode se dizer que é um sistema gerador de linguagens e de reescrita, uma forma de representar linguagens, e até mesmo um dispositivo formal para especificar de modo preciso e finito uma linguagem possivelmente infinita.

As gramáticas regulares (GR) fornecem vários modos de determinar uma linguagem regular. Os autômatos finitos apresentados nos capítulos anteriores permitem a especificação de uma linguagem através de um reconhecedor para a mesma, já as gramáticas regulares permitem especificação através de um gerador de linguagem, ou seja, mediante a uma gramática regular, aponta como gerar todas, e apenas, as palavras de uma linguagem regular (VIEIRA, 2006).

Conforme Ramos (2009) as GR pertencem a classe do tipo 3 na hierarquia de Chomsky, ou seja, das gramáticas lineares á direita ou á esquerda. A hierarquia consiste na classificação das linguagens em quatro classes distintas denominadas tipos 0, 1, 2 e 3, onde o relacionamento entre elas pode ser resumido da seguinte forma:

a) toda gramática do tipo 3 é também do tipo 2;
b) nem toda gramática do tipo 2 é também do tipo 1. Tipo 1 são apenas aquelas que não possuem produções $\alpha \rightarrow \beta$ em que $\beta = \varepsilon$;
c) toda gramática do tipo 1 é também do tipo 0.

A definição das classes é feito por intermédio das diferenças nas restrições aplicadas ao formato das produções ($\alpha \rightarrow \beta$) das gramáticas que geram as linguagens, como segue:

a) tipo 0 ou gramática irrestrita: como o próprio nome sugere, são gramáticas que não possuem nenhuma restrição em suas produções. A linguagem gerada por essa classe gramatical é chamada de linguagem recursivamente enumerável, irrestrita, ou do tipo 0;
b) tipo 1 ou sensíveis ao contexto - apresenta restrição no comprimento das formas sentenciais, onde o comprimento da cadeia do lado direito de cada produção seja no mínimo igual ao comprimento da cadeia do lado esquerdo, exceto para produção $S \rightarrow \varepsilon$, onde S não pode aparecer no lado direito de alguma produção:
   - $\alpha \in V^* NV^*$
   - $\beta \in V^*$
   - $| \beta | \geq | \alpha |$.

   A linguagem gerada por essa classe gramatical é chamada linguagem sensível ao contexto ou tipo 1;
c) tipo 2 ou livre de contexto - suas produções devem possuir apenas um símbolo não-terminal em seu lado esquerdo, e uma combinação qualquer de símbolos terminais e não-terminais no lado direito:
   - $\alpha \in N$
   - $\beta \in V^*$.

   A linguagem gerada por essa classe gramatical é chamada de linguagem livre do contexto ou tipo 2;
d) tipo 3 ou gramática regular - é das gramáticas lineares que apresenta as regras de produção especificas para gerar uma linguagem regular ou do tipo 3, sendo a classe de linguagens mais simples da hierarquia. As regras dessa classe serão apresentadas a seguir.

De acordo com Menezes (2005) para definir uma gramática regular algumas formas de restrições nas regras de produções devem ser seguidas. Essas regras são estabelecidas pelas gramáticas lineares, que apresentam 4 formas de restrições em suas produções:

a) Gramática Linear à Direita (GLD) - apresenta todas as regras de produções da seguinte forma:

$$A \rightarrow wB \quad \text{ou} \quad A \rightarrow w$$

b) Gramática Linear à Esquerda (GLE) - apresenta todas as regras de produções da seguinte forma:

$$A \rightarrow Bw \quad \text{ou} \quad A \rightarrow w$$

c) Gramática Linear Unitária à Direita (GLUD) - apresenta todas as regras de produções como a GLD, acrescentando:

$$|\, w \,| \leq 1$$

d) Gramática Linear Unitária à Esquerda (GLUE) - apresenta todas as regras de produções como a GLE, acrescentando:

$$|\, w \,| \leq 1$$

As gramáticas regulares geram apenas linguagens regulares e, vice-versa. Com isso através de uma GR podemos gerar um AF que reconhece a linguagem da GR, e através de um AF podemos gerar a GR referente à linguagem que o AF reconhece. Nos próximos dois subcapítulos serão apresentados os teoremas para aplicar essas transformações.

3.1 GERAÇÃO DE UM AUTÔMATO FINITO A PARTIR DE UMA GRAMÁTICA REGULAR

Para provar que uma linguagem gerada por uma gramática regular é uma linguagem regular devemos gerar o autômato finito que a reconheça (MENEZES, 2005).

Conforme Ramos (2009) para efetuar essa conversão, devemos ter como entrada uma GR, onde é aplicado um algoritmo que retorna um AF M tal que ACEITA(M) = GERA(G). O algoritmo segue:

a) conjunto de estados: Cada símbolo não terminal da gramática corresponde a um estado em M. O estado inicial de M é a raiz da gramática (S), e o estado final é Z, que é um novo estado que deve ser acrescentado ao conjunto de estados;
b) alfabeto de entrada: O alfabeto de entrada Σ de M é o alfabeto de símbolos terminais T de G;
c) função de transição (δ):

   $\delta \leftarrow \emptyset$;

   Para cada regra de produção em P de G, e conforme seu tipo:

- se X → aY então δ ← δ(X, a) →Y,
- se X → a então δ ← δ(X, a) → Z,
- se X → ε então X é terminal e não gera transição.

Segue abaixo um exemplo da construção de um AFND a partir da GR = ({S, A}, {a, b}, P, S), onde P é:

$$S \rightarrow aA \mid bA \mid \varepsilon$$

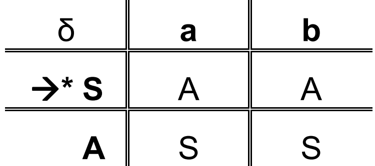
$$A \rightarrow aS \mid bS$$

O AF que reconhece a linguagem gerada pela gramática regular acima é:

$$M = (\{a, b\}, \{S, A\}, \delta, S, S).$$

Observa-se que não foi necessária a inclusão do novo estado Z no conjunto de estados do AF, pois não teve ocorrência da regra de produção X → a. A tabela 6 é apresenta as funções de transição (δ) do AF M gerado seguindo os passos do algoritmo apresentado anteriormente. Já na figura 6 mostra o AF M na forma de diagrama de transição.

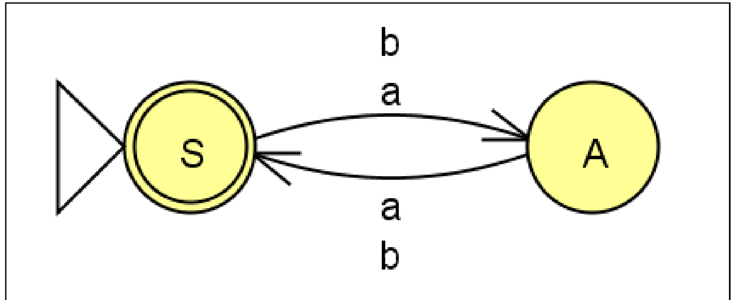
Tabela 5 - Transições geradas a partir das produções.



δ	**a**	**b**
→* S	A	A
A	S	S

Fonte: Do autor.

Figura 6 - AF construído a partir da uma GR.



Fonte: Do autor.

3.3 GERAÇÃO DE UMA GRAMÁTICA REGULAR A PARTIR DE UM AUTÔMATO FINITO

Se uma linguagem L é regular, então existe uma gramática regular G que gera L, ou seja, se L é uma linguagem regular, então a linguagem reconhecida por determinado AFD M = (Σ, Q, δ, q0, F) é igual L. Para demonstrar isso, devemos

construir uma gramática regular a partir do autômato finito onde GERA(G) = ACEITA(M) (MENEZES, 2005).

Conforme Ramos (2009) para efetuar essa conversão, devemos ter como entrada um autômato finito M, onde é aplicado um algoritmo que retorna uma GR tal que L(G) = L(M). O algoritmo segue:

a) definição dos conjuntos de símbolos não terminais: os símbolos não terminais de G são os estados M. O símbolo inicial da gramática será q0;
b) Definição do alfabeto de entrada: O alfabeto T de G é o próprio alfabeto de entrada Σ de M;
c) Produções:

   $P \leftarrow \emptyset$

   Para cada transição de δ (função de transição) do AF M, e conforme o tipo das transições de M :

   - se $\delta(X , a) = Y$, então $P \leftarrow \{ X \rightarrow aY \}$;
   - se $\delta(X , a) = Y$, e $Y \in F$ então $P \leftarrow \{ X \rightarrow a \}$.

   $F \leftarrow \emptyset$

   Para cada elemento de Q (conjunto de estados) do AF M:

   - Se $X \in F$, então $P \leftarrow \{ X \rightarrow \varepsilon \}$.

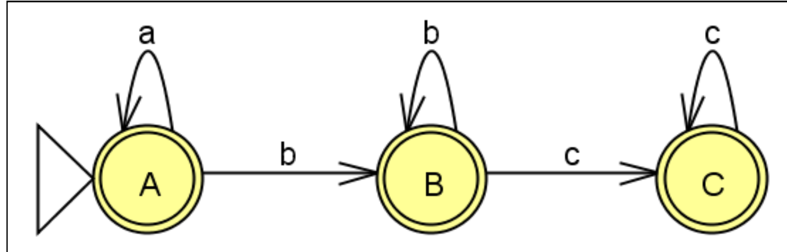
Segue abaixo um exemplo da construção de uma GR a partir do AFD = ({a, b, c}, {A, B, C}, δ, A, {A, B, C}). A função de transição δ do AFD é apresentada na tabela 6, e o diagrama de transição na figura 7.

Tabela 6 - Transições do AFD para gerar a GR.

δ	**a**	**b**	**c**
→* A	A	B	-
*** B**	-	B	C
*** C**	-	-	C

Fonte: Do autor.

Figura 7 - Diagrama de transição do AFD para gerar a GR.



Fonte: Do autor.

Após aplicar o algoritmo de conversão a gramática regular construída que gera a linguagem reconhecida pelo autômato finito acima é:

GR = ({A, B, C}, {a, b, c}, P, A).

Na tabela 7 é dado P (produções) da GR geradas de acordo com as transições do AFD em questão. Segue a gramática regular com os símbolos não terminais e suas respectivas produções:

A → aA | bB | a | b | ε

B → bB | cC | b | c | ε

C → cC | c | ε

Tabela 7 - Produções geradas a partir das transições.

δ	**P**
δ(A, a) = A	A → Aa | a
δ(A, b) = B	A → bB | b
δ(B, b) = B	B → bB | b
δ(B, c) = C	B → cC | c
δ(C, c) = C	C → cC | c
Q	**P**
A ∈ F	A → ε
B ∈ F	B → ε
C ∈ F	C → ε

Fonte: Do autor.

4 AFLAB

O AFLAB é um aplicativo inicialmente desenvolvido em um Trabalho de Conclusão de Curso de Ciência da Computação, da Universidade do Extremo Sul Catarinense. O projeto tinha como objetivo o desenvolvimento de uma ferramenta de auxilio no aprendizado dos alunos nas disciplinas de Linguagens Formais, Compiladores e Teoria da Computação do curso de Ciência da Computação da UNESC, onde disponibilizaria um ambiente de manipulação e criação de autômatos finitos.

O aplicativo AFLAB foi desenvolvido na linguagem de programação Object Pascal, onde utilizou como ambiente de desenvolvimento o software Borland® Delphi™ Enterprise versão 7.0. A linguagem foi escolhida pelo fato de ser bastante utilizada como objeto de estudo no meio acadêmico da época, possibilitando estudos e alterações no código fonte pelos alunos, e facilitando a implementação de novos módulos futuramente. Outro fato que influenciou na escolha foi à capacidade de criar softwares leves, e que não utilizem a memória do computador em excesso (GAIDZINSKI, 2007).

Primeiramente, o AFLAB foi desenvolvido por Marco Aurélio Gaidzinski, que disponibilizou dois módulos para reconhecimento de sentenças a partir de autômatos finitos determinísticos e não determinísticos, onde um módulo era para definição de AF na forma gráfica e outro na forma tabular, permitindo a visualização nas duas formas (GAIDZINSKI, 2007).

Gaidzinski (2007) definiu uma estrutura única para o armazenamento dos AF, independente do autômato ser determinístico ou não determinístico, onde foi dividida em três classes:

a) Testados: responsável pelo armazenamento dos estados, onde tem como atributos o nome, marcação de estado inicial ou final, e as coordenadas referentes à posição do estado na tela;
b) Talfabeto: contém armazenado o alfabeto de símbolo de entrada do AF;
c) TConexao: contém as informações referentes a cada transição do AF, onde tem como atributos o estado origem, o símbolo de entrada que deve fazer parte do alfabeto armazenado em Talfabeto, e por ultimo o

estado destino. Os estados origem e destino devem pertencer ao conjunto de estados armazenados em Testados.

Essa estrutura de armazenamento é a base para o desenvolvimento das funcionalidades de cada módulo, onde posteriormente novos módulos foram implementados de acordo com os trabalhos de conclusão de curso desenvolvidos em cima do AFLAB.

Teixeira (2008) em seu trabalho de conclusão de curso desenvolveu dois novos módulos, um para definições de gramáticas regulares e outro para definições de expressões regulares, onde cada um, de acordo com a definição, gera o autômato finito referente. Já Oliveira (2008) disponibilizou mais dois módulos, um para a máquina de Moore e outro a de Mealy, onde apresentam como saída áudios, imagens e textos, associados aos estados (Moore) e as transições (Mealy). E por último, Serafin (2009) também em seu TCC incluiu os módulos para transformação de AFND em AFD, minimização de autômatos finitos, e para geração de uma gramática regular a partir de um autômato finito determinístico.

4.1 ALGORITMOS UTILIZADOS NOS MÓDULOS DO AFLAB

Neste capítulo é apresentado os códigos fontes dos algoritmos implementados nos módulos do aplicativo AFLAB. Estes algoritmos foram colocados em pratica através dos estudos levantados durante os quatros trabalhos de conclusão de curso desenvolvidos sobre o AFLAB, e servirão como base no desenvolvimento do aplicativo web, de manipulação de autômatos finitos e gramáticas regulares.

Na figura 8 pode ser verificado as declarações das classes Testados, Talfabeto e Tconexão, que representam a estrutura de armazenamento de AF criada por Gaidzinski (2007).

Figura 8 - Estrutura de Armazenamento de AF.

```
type
  Testados = class
  nome_estado :string;
  estado_final: Boolean;
  Estado_inicial: Boolean;
  PosX, PosY: integer;
  procedure  NovoEstado(Rotulo,  Efinais,  Einicial:  string  ;  X,  Y:
   integer);
  procedure ApagaEstados;
  procedure DefineEstadoFinal(EstadosFinais: string);
  procedure DefineEstadoInicial(EstadoInicial: string);
  function ContaEstados: integer;
  function VerificaEstadoRepetido(Nome: string): Boolean;
  procedure RemoveEstado(Posicao: integer);
  procedure AtualizaEstadoFinal(Finais: string);
end;
  Talfabeto = class
  letra_alfabeto :string;
  procedure NovoAlfabeto(Rotulo: string);
  procedure ApagaAlfabeto;
  function VerificaAlfabetoRepetido(Simbolo: string): Boolean;
end;
  Tconexao = class
  estado_origem, estado_destino, Caractere :string;
  procedure NovaConexao(Origem, Destino, Rotulo: string);
  procedure ApagaConexoes;
  procedure MostraConexoes(num_conexoes: integer);
  procedure RemoveConexoes(NomeEstado: string);
  procedure ExcluiConexao(PosVet: integer);
  function TestaSentenca(EstadoInicial, Sentenca: string): Boolean;
  function TestaEstadoFinal(Estado: string): Boolean;
  function BuscaProximoEstado(Estado, Alfabeto: string): string;
  function ContaConexoes: integer;
  function   VerificaConexoesRepetidas(EOrigem,   EDestino,   Simbolo:
string): Boolean;
end;
```

Fonte: Gaidzinski (2007, p.42).

Para o armazenamento dos dados foi utilizado os vetores dinâmicos, devido aos estados, símbolos de entrada e transições, não ficarem limitados a um número fixo de dados na definição do AF (GAIDZINSKI, 2007). Segue na figura abaixo (figura 9) a declaração desses vetores:

Figura 9 - Tipos declarados de Vetores.

```
var
 Estados: array of Testados;
 Alfabeto: array of Talfabeto;
 Conexao: array of Tconexao;
```

Fonte: Gaidzinski (2007, p.43).

De acordo com Gaidzinski (2007) para o reconhecimento de sentenças foi implementado um algoritmo que foi dividido em três funções (figuras 10, 11, 12), como segue:

a) TestaSentenca: A partir do símbolo inicial da sentença e em conjunto com a função BuscaProximoEstado, percorre a sentença até chegar ao final;
b) BuscaProximoEstado: recebe como parâmetro o símbolo de entrada e o conjunto de estados, e retorna o conjunto de estados acessíveis;
c) TestaEstadoFinal: recebe o ultimo conjunto de estados retornado por BuscaProximoEstado e verifica se contém algum estado final, retornando TRUE se tiver e FALSE se não tiver.

Figura 10 - Função testa sentença.

```
function   Tconexao.TestaSentenca(EstadoInicial,   Sentenca:   string):
Boolean;
var i, c, cont: integer;
S, S2, S3: string;
begin
  Result := False;
  cont := 0;
  if (length(Conexao) <= 0) then
  begin
     Exit;
  end;
  S2 := Trim(EstadoInicial) + ',';
  S3 := '';
  while Length(Sentenca) >  0 do
  begin
     S := Trim(Copy(Sentenca, 1, 1));
     Delete(Sentenca, 1, 1);
     c := Pos(',', S2);
     if ((EstadoInicial + ',')= S2) and (cont = 0) then
     begin
        cont := 1;
        S2 := BuscaProximoEstado(Copy(S2, 1, c -1), S);
        if Length(Sentenca) = 0 then
        begin
          Result := TestaEstadoFinal(Trim(S2));
          Exit;
        end;
        S := Trim(copy(Sentenca, 1, 1));
        //Delete(Sentenca, 1, 1);
     end;
     c := Pos(',', S2);
     while c > 0 do //Estados
     begin
      if BuscaProximoEstado(Copy(S2, 1, c - 1), S) <> '' then
        S3 :=  Trim(S3) + BuscaProximoEstado(Copy(S2, 1, c - 1), S);
      delete(S2, 1, c);
      c := Pos(',', S2);
     end;
     S2 := Trim(S3);
     S3 := '';
  end;
  Result := TestaEstadoFinal(Trim(S2));
end;
```

Fonte: Gaidzinski (2007, p.46).

Figura 11 - Função testa estado final.

```
function Tconexao.TestaEstadoFinal(Estado: string): Boolean;
var i, c: integer;
S: string;
begin
  c := Pos(',', Estado);
  while c > 0 do
  begin
    S := Trim(Copy(Estado, 1, c - 1));
    Delete(Estado, 1, c);
    for i := 0 to NEstados - 1 do
    begin
      if (S = Trim(Estados[i].nome_estado)) and
(Estados[i].estado_final) then
      begin
        Result := True;
        Exit;
      end
      else
        Result := False;
    end;
    c := Pos(',', Estado);
  end;
end;
```

Fonte: Gaidzinski (2007, p.47).

Figura 12 - Função busca próximo estado.

```
function    Tconexao.BuscaProximoEstado(Estado,    Alfabeto:    string):
string;
var i: integer;
S: string;
begin
  S := '';
  for i := 0 to NConexoes - 1 do
  begin
     if      (Conexao[i].estado_origem      =      Estado)           and
(Conexao[i].Caractere = Alfabeto) then
        S := Trim(S) + Conexao[i].estado_destino + ',';
  end;
  Result := S;
end;
```

Fonte: Gaidzinski (2007, p.47).

Conforme Teixeira (2008) para gerar um autômato finito a partir de uma gramática regular foi implementado o código fonte (figura 13), onde a função

determinaPalavras recebe a parte esquerda e direita das produções e retorna três palavras separadas por virgula para serem tratadas. Cada produção vai gerar uma transição e assim vai sendo montado o autômato finito.

Figura 13 - Código para geração do autômato finito a partir da gramática regular

```
for i := 0 to strgrid_gr1.RowCount - 1 do
  if strgrid_gr1.Cells[0,i] <> ''
then begin
  palavra0 := '';
  palavra1 := '';
  palavra2 := '';
  achou1virg := false;
  achou2virg := false;

  palavra := determinaPalavras(strgrid_gr1.Cells[0,i],
             strgrid_gr1.Cells[2,i]);

  for j := 1 to Length(palavra) do
  begin
    if (palavra[j] <> ',')  and
       (achou1virg = false) and
       (achou2virg = false)
    then palavra0 := palavra0 + palavra[j]
    else if palavra[j] <> ','
         then if achou2virg = false
              then palavra1 := palavra1 + palavra[j]
              else palavra2 := palavra2 + palavra[j];

    if palavra[j] = ','
    then begin
      if achou1virg = false
      then achou1virg := true
      else achou2virg := true;
    end;
  end;

  if (palavra1 = 'î') or
     (palavra1 = 'Î')
  then begin
    if (estFinais <> '')
    then estFinais := estFinais + ',';

    estFinais := estFinais + strgrid_gr1.Cells[0,i];
  end
  else if (palavra2 = '') then
  begin
    novoestado := 'Z';

    estAutomato := estAutomato + ',' + novoestado;

    if (estFinais <> '')
    then estFinais := estFinais + ',';

     estFinais := estFinais + novoestado;

    if ListaConexoes.VerificaConexoesRepetidas(Trim(palavra0),
       Trim(novoestado), Trim(palavra1)) = false
    then ListaConexoes.NovaConexao(Trim(palavra0), Trim(novoestado),
         Trim(palavra1));
  end
  else if ListaConexoes.VerificaConexoesRepetidas(Trim(palavra0),
          Trim(palavra2), Trim(palavra1)) = false
       then ListaConexoes.NovaConexao(Trim(palavra0), Trim(palavra2),
            Trim(palavra1));
   if (palavra1 <> '') and //Cria o alfabeto do autômato
      (palavra1 <> 'î') and
      (palavra1 <> 'Î')
   then ListaAlfabeto.NovoAlfabeto(palavra1);
end;
for i := 1 to Length(estAutomato) do //Cria os estados do autômato
begin
  if estAutomato[i] <> ','
  then estado := estado + estAutomato[i];

  if (estAutomato[i] = ',') or
     (i = Length(estAutomato))
  then begin
    posY := ListaEstados.ContaEstados+1;

    if (posY mod 2 = 0)
    then posY := posY + 30;

    ListaEstados.NovoEstado(Trim(estado), Trim(estFinais),
                            Trim(ed_GrSimbIni.Text),
                            ValorX(ListaEstados.ContaEstados+1),
                            ValorY(posY));

    estado := '';
  end;
end;
```

Fonte: Teixeira (2008, p.70).

Segundo Oliveira (2008), os seus módulos de autômatos finitos com saída (AFS) não contemplam os AFND, restringindo assim, a simulação de saídas para apenas AFD. Então para gerar AFS primeiramente devemos criar o AF na forma tabular ou gráfica, e após clicar na opção Mealy ou Moore que efetuará verificações como se o autômato está criado, se existem transições e se o autômato é determinístico. Se todas as verificações forem atendidas, o sistema monta uma grade com os dados do AF, onde uma coluna corresponde a todos os estado(Moore) ou todas as transição(Mealy), e outra coluna para o preenchimento das saídas pelo usuário. A figura 14 apresenta a função que processa esses dados e cria a grade.

Figura 14 - Parte do código que monta a grade (Simulação Mealy).

```
for i := 0 to NConexoes - 1  do
begin
  if  (NConexoes >= 0) and (Conexao[i].ContaConexoes <= NConexoes)  then
  begin
    cds_mealy.Append;
    cds_mealy['Campo1']:='('+Conexao[i].estado_origem+','+Conexao[i].Caractere+') = '+Conexao[i].estado_destino;
    cds_mealy.Post;
  end;
end;
```

Fonte: Oliveira (2008, p.53).

Com a grade gerada deve ser escolhido o tipo de saída, e após relacionar as saídas aos estados ou transições. Para simular a saída deve ser informada a sentença e após clicar em simular, onde será verificado qual o tipo da saída escolhida e se existe uma sentença informada. A função de imagem, de som ou de texto é chamada de acordo com o tipo de saída escolhido (OLIVEIRA, 2008).

Figura 15 - Parte do código que testa tipo de saída e sentença informada (Simulação Mealy).

```
procedure Tfrm_principal.bt_saida_mealyClick(Sender: TObject);
var
teste: boolean;
begin
teste := true;
mm_tex_mealy.Clear;

if(Length(ed_sent_mealy.Text) > 0) then //testa se a sentença foi informada
begin
  //se a saida for texto
  if rg_saida_mealy.ItemIndex = 0 then
  begin
    teste := SaidasMealyTexto(ed_ei_mealy.Text, ed_sent_mealy.text, 1000);
  end
```

Fonte: Oliveira (2008, p.54).

Figura 16 - Parte do código que define a função de acordo com a saída (Simulação Mealy).

```
//se a saida for imagem
if rg_saida_moore.ItemIndex = 1 then
begin
  teste := SaidasMooreImagem(ed_ei_moore.Text, ed_sent_moore.text);
end
else
  //se a saida for som
  if rg_saida_moore.ItemIndex = 2 then
  begin
    mp_moore.Enabled := true;
    mm_tex_moore.Visible := true;
    mm_tex_moore.Clear;
    mm_tex_moore.Lines.Add('ORDEM DOS SONS...');
    teste := SaidasMooreTexto(ed_ei_moore.Text, ed_sent_moore.text, 0);
    if (teste = false) then
      exit
    else
    begin
      sb_play_moore.Enabled := true;
      sb_play_moore.Down := true;
      sb_next_moore.Enabled := true;
      mp_moore.Enabled := true;
      teste := SaidasMooreSom(ed_ei_moore.Text, ed_sent_moore.text);
    end;
```

Fonte: Oliveira (2008, p.55).

Para reproduzir as saídas em Mealy foram criadas três funções que são apresentadas nas figuras 17, 18, 19.

Figura 17 - Função que compara o símbolo de entrada com da transição (Mealy).

```
cds_mealy.First; //marca a primeira posicao da grid
while not cds_mealy.Eof do //enquanto nao chegar ao fim da grid
begin
  for i := 0 to NConexoes -1 do
  begin
    if ((cds_mealy['Campo1']+',') = ('('+EstadoInicial + ','+S+') = '+S2))  then
    begin
      mm_tex_mealy.Lines.Add(cds_mealy['Campo2']); //escreve no memo a saida
      sleep(tempo); //da uma pausa em milisegundos...
      teste := 1;
    end;//fecha o if
    if (teste = 1) then
      break;
end;//fecha o for
cds_mealy.Next; //passa grid para pxoxima celula
break;
```

Fonte: Oliveira (2008, p.56).

Figura 18 - Função de saída com imagem (Mealy).

```
c := Pos(',', S2);
teste := 0;
while c > 0 do //equanto tiver estados
begin
  if ListaConexoes.BuscaProximoEstado(Copy(S2, 1, c - 1), S) <> '' then
  begin
    S3 :=  Trim(S3) + ListaConexoes.BuscaProximoEstado(Copy(S2, 1, c - 1), S);
    cds_mealy.First; //marca a primeira posicao da grid
    while not cds_mealy.Eof do //enquanto nao chegar ao fim da grid
    begin
      for i := 0 to NConexoes -1 do
      begin
        S4 := Trim(copy(S2, 1, c-1));
        if (cds_mealy['Campo1']+',') = ('('+S4 + ','+S+') = '+S3) then
        begin
          image_mealy.Picture.LoadFromFile( cds_mealy['Campo2'] );
          image_mealy.Show;
          application.ProcessMessages;
          sleep(1000);
          teste := 1;
        end;//fecha if
        if (teste = 1) then
          break;
      cds_mealy.Next;//passa grid para pxoxima celula
      end;//fecha o for
    break;
    end;//fecha o while da grid
  break;
  end;//fecha o if da proxima conexao
Delete(S2, 1, c);
c := Pos(',', S2);
end;//fecha o while do c>0
S2 := Trim(S3);
S3 := '';
```

Fonte: Oliveira (2008, p.57).

Figura 19 - Função de saída com som (Mealy).

```
cds_mealy.First; //marca a primeira posicao da grid
while not cds_mealy.Eof do //enquanto nao chegar ao fim da grid
begin
  for i := 0 to NConexoes -1 do
  begin
    S4 := Trim(copy(S2, 1, c-1));
    if ((cds_mealy['Campo1']+',') = ('('+S4 + ','+S+') = '+S3)) then
    begin
      if(mp_mealy.Enabled = true) then
      begin
        mp_mealy.FileName := cds_mealy['Campo2'];
        mp_mealy.Open;
        mp_mealy.Play;
        while mp_mealy.Mode = mpPlaying do
        begin
          lb_toc_mealy.Caption:='Tocando música ' + cds_mealy['Campo2'];
          application.ProcessMessages;
        end; //fecha o while
      end//fecha o if do enabled
      else
        exit;
      teste := 1;
    end;//fecha if
    if (teste = 1) then
      break;
  cds_mealy.Next;//passa grid para pxoxima celula
  end;//fecha o for
break;
end;//fecha o while da grid
```

Fonte: Oliveira (2008, p.58).

A figura 20 apresenta a função de saída de texto na máquina de Moore, que também foi dividida em três funções, onde todas seguem a mesma lógica de implementação.

Figura 20 - Função de saída com texto (Moore).

```
if ListaConexoes.BuscaProximoEstado(Copy(S2, 1, c - 1), S) <> '' then
begin
  S3 := Trim(S3)+ListaConexoes.BuscaProximoEstado(Copy(S2, 1, c - 1), S);
  cds_moore.First; //marca a primeira posicao da grid
  while not cds_moore.Eof do //enquanto nao chegar ao fim da grid
  begin
    S4 := Trim(copy(S2, 1, c - 1));
    for i := 0 to NConexoes -1 do
    begin
      if ((cds_moore['Campo1']) = (S4))then
      begin
        mm_tex_moore.Lines.Add(cds_moore['Campo2']); //escreve no memo
        sleep(tempo);
        teste := 1;
      end;//fecha if
      if (teste = 1) then
        break;
      cds_moore.Next; //passa grid para pxoxima celula
    end;//fecha o for
  break;
  end;//fecha o while da grid
break;
end;//fecha o if da proxima conexao
```

Fonte: Oliveira (2008, p.59).

Agora tendo um AFND e necessitar na forma de AFD deve ser aplicado o algoritmo de transformação, sendo assim, Serafin (2009) desenvolveu um módulo que aplica essa transformação. De acordo com conteúdos pesquisados foram criadas duas classes (figura 21) para armazenamento de estados e transições, como seguem:

a) Testados_tranf – Armazenamento de estados do AFD criado, tendo como atributos o nome e a identificação se é estado inicial ou final;
b) Tconexao_tranf – Armazenamento de transições, tendo como atributos o estado origem, estado destino e símbolo de entrada.

Figura 21 - Declaração das classes de armazenamento do AFD.

```
type

    Testados_tranf = class
    nome_estado_tranf: string;
    estado_final: Boolean;
    Estado_inicial: Boolean;
    procedure NovoEstado_tranf(Rotulo: string; EFinais, Einicial: Boolean);
    function ContaEstados_tranf: integer;
    procedure ApagaEstados_tranf;
end;

    Tconexao_tranf = class
    estado_origem, estado_destino, Caractere :string;
    procedure NovaConexao_tranf(Origem, Destino, Rotulo: string);
    procedure ApagaConexoes_tranf;
    function ContaConexoes_tranf: integer;
    procedure ExcluiConexao_tranf(PosVet: integer);
end;
```

Fonte: Serafin (2009, p.50).

Pelos mesmos motivos apresentados por Gaidzinski (2007), foram utilizados os vetores dinâmicos (figura 22) para o armazenamento dos estados e transições (SERAFIN, 2009).

Figura 22 - Tipos declarados.

```
var

  Estados_tranf: array of Testados_tranf;
  Conexao_tranf: array of Tconexao_tranf;
```

Fonte: Serafin (2009, p.50).

De acordo com Serafin (2009) o algoritmo de transformação foi implementado na *procedure trasformação*. Inicialmente o algoritmo determina qual o primeiro estado do AFD (figura 24), sendo assim, o estado inicial do AFND passa a ser o inicial do AFD. A próxima etapa é apresentada na figura 23, que consiste em verificar se as transições foram determinadas para cada estado do AFD, onde em caso de negativo é criado um novo estado em AFD, e para cada símbolo de entrada é feito uma união das transições dos referentes estados do AFND, que são incluídas no AFD junto com a identificação de estado final ou não. Essa inclusão das uniões das transições é demonstrada na figura 25.

Figura 23 - Identificação se a transição foi determinada.

```
while (m<ListaEstados_tranf.ContaEstados_tranf+1)do begin
  for n:=0 to ListaEstado2.ContaEstados_tranf -1 do begin
    for j:=0 to ListaAlfabeto.ContaAlfabeto-1 do begin
      for p:=0 to  ListaConexoes2.ContaConexoes_tranf-1 do begin
        if (Conexao_tranf[p].estado_origem=Estados_tranf[n].nome_estado_tranf)
        and(Conexao_tranf[p].Caractere=Alfabeto[j].letra_alfabeto) then begin
            status3[h]:=Conexao_tranf[p].estado_destino+',';
              for x:=0 to ListaEstados.ContaEstados -1 do begin
                  if (Estados[x].nome_estado=Conexao_tranf[p].estado_destino) then begin
                      if (Estados[x].estado_final=True) then begin
                          status6[y]:=status6[y]+1;
                      end;

                      if (Estados[x].estado_final=False)or(Estados[x].Estado_inicial=True) then begin
                          status6[y]:=status6[y]+0;
                      end;
                  end;
              end;
            h:=h+1;
        end;
      end;
      ok2:=True;
      while (ok2<>false) do begin
            ok2:=False;
            indice2:=0;
            while (indice2<h -1) do begin
                if CompareStr(status3[indice2], status3[indice2 + 1]) > 0 then begin
                    valor4 := status3[indice2];
                    status3[indice2]:= status3[indice2+1];
                    status3[indice2+1]:= valor4;
                    ok2 := True;
                end;
                indice2:=indice2+1;
            end;
      end;

      for i:=0 to h-1 do begin
            if (status3[i]<>'') then begin
                status2[a]:=status2[a]+status3[i];
            end;
      end;


    a:=a+1;
    h:=0;
    y:=y+1;

 for h:=0 to a -1 do begin
      indicador:=0;
      for g:=0 to ListaEstados_tranf.ContaEstados_tranf-1 do begin
            if (status2[h]=Estados_tranf[g].nome_estado_tranf) or (status2[h]= '') then begin
                indicador:=1;
            end;

            if (Estados_tranf[g].Estado_inicial=True)and (status2[h]=Estados_tranf[g].nome_estado_tranf+',') then begin
              indicador:=1;
            end;
      end;
```

Fonte: Serafin (2009, p.52).

Figura 24 - Definição do estado inicial no AFD.

```
for i:=0 to ListaEstados.ContaEstados -1 do begin
    if (Estados[i].Estado_inicial=True) then begin
        ListaEstado2.NovoEstado_tranf(Estados[i].nome_estado,Estados[i].estado_final,Estados[i].Estado_inicial);
        for g:=0 to ListaConexoes.ContaConexoes -1 do begin
            if (Conexao[g].estado_origem = Estados[i].nome_estado) then begin
             ListaConexoes_tranf.NovaConexao_tranf(Conexao[g].estado_origem,
             Conexao[g].estado_destino, Conexao[g].Caractere);
            end;
        end;
    end;
end;
```

Fonte: Serafin (2009, p.51).

Figura 25 - Inclusão do estado no AFD.

```
 if (indicador=0) then begin
          if (status6[h]>0) then begin
              ListaEstado2.NovoEstado_tranf(status2[h],True,False);
          end;
          if (status6[h]=0) then begin
              ListaEstado2.NovoEstado_tranf(status2[h],False,False);
          end;
```

Fonte: Serafin (2009, p.53).

A figura 26 mostra o código da terceira etapa do processo, que tem o objetivo de identificar a transição de cada estado incluído em *Testados_tranf* e armazena-las em *Tconexao_tranf,* utilizando uma ordenação para evitar duplicidades. Ao fim desta etapa, se solicitado pelo usuário, os nomes dos estados são renomeados automaticamente pelo sistema para facilitar a identificação, terminando assim o algoritmo de transformação de AFND em AFD (SERAFIN, 2009).

Figura 26 - Inclusão das transições dos estados no AFD.

```
        while i > 0 do begin
             S2:= Trim(Copy(S, 1, i - 1)); // Copia desde 1 até a virgula - 1
             Delete(S, 1, i); // Apaga desde 1 até a virgula
             if Length(S2) > 0 then  begin // Se copiou alguma coisa
                  for i:=0 to ListaConexoes.ContaConexoes-1 do begin
                      cont:=0;
                      if (Conexao[i].estado_origem=S2) then begin
                          for j:=0 to ListaConexoes_tranf.ContaConexoes_tranf -1 do begin
                                if ((Conexao_tranf[j].estado_destino)=(Conexao[i].estado_destino))
                                and(Conexao_tranf[j].estado_origem=status2[h])
                                and(Conexao_tranf[j].Caractere=Conexao[i].Caractere) then begin
                                    cont:=1;
                                end;
                          end;
                          if (cont=0) then begin
                              ListaConexoes2.NovaConexao_tranf(status2[h],Conexao[i].estado_destino,Conexao[i].Caractere);
                          end;
                      end;
                  end;
             end;
        i:= Pos(',', S); // Pega a posicao das proxima virgula

        end;
    ok:=True;
    while (ok<>false) do begin
        ok:=False;
        indice:=0;
   while (indice<ListaConexoes_tranf.ContaConexoes_tranf -1) do begin

         if CompareStr(Conexao_tranf[indice].estado_destino, Conexao_tranf[indice + 1].estado_destino) > 0 then
             begin
                   //ShowMessage('ordenar:');

                   valor1 := Conexao_tranf[indice].estado_origem;
                   valor2 := Conexao_tranf[indice].estado_destino;
                   valor3 := Conexao_tranf[indice].Caractere;
                   Conexao_tranf[indice].estado_origem:= Conexao_tranf[indice+1].estado_origem;
                   Conexao_tranf[indice].estado_destino:= Conexao_tranf[indice+1].estado_destino;
                   Conexao_tranf[indice].Caractere:= Conexao_tranf[indice+1].Caractere;
                   Conexao_tranf[indice+1].estado_origem:= valor1;
                   Conexao_tranf[indice+1].estado_destino:=valor2;
                   Conexao_tranf[indice+1].Caractere:=valor3;
                   ok := True;

               end;
     indice:=indice+1;
     end;
end;
```

Fonte: Serafin (2009, p.54).

Ainda seguindo os algoritmos de Serafin (2009), temos a implementação do módulo de minimização de autômatos finitos, que de acordo com o que já foi abordado anteriormente, consiste na eliminação dos estados inacessíveis, mortos e equivalentes de um AFD.

Para eliminação dos estados inacessíveis foi criado a *procedure Elimina_estados_inacessiveis*, demonstrado na figura 27, onde é dado como inacessível o estado que não for inicial e destino de nenhum estado. A eliminação é

feita através do método *RemoveConexoes* da classe *Tconexao*, e após é inserido em um vetor para posteriormente ser eliminado do AF (SERAFIN, 2009).

Figura 27 - Eliminação de estados inacessíveis.

```
procedure Tfrm_minimizacao.Elimina_estados_inacessiveis;

var
h, i, C, R, j, n, status, status1, status2 :integer;
S , T:string;
ListaElimina : TElimina;

begin
status:=0;
status:=0;
NElimina:=0;
for i:=0 to ListaEstados.ContaEstados-1 do
    begin
  for j := 0 to ListaConexoes.ContaConexoes -1     do
     begin
          if (Estados[i].nome_estado <> Conexao[j].estado_destino)and(Estados[i].estado_inicial=False) then begin
          status1:=status1+1;
          end;
          if (Estados[i].nome_estado = Conexao[j].estado_destino)and(Estados[i].nome_estado = Conexao[j].estado_origem)
          and(Estados[i].estado_inicial=False)then begin
          status1:=status1+1;
          end;
     end;
     if (status1=ListaConexoes.ContaConexoes) then begin
        ListaConexoes.RemoveConexoes(Estados[i].nome_estado);
        ListaElimina.NovoElimina(Estados[i].nome_estado);
     end;

   status1:=0;
end;
status:=0;
        for i:=0 to ListaElimina.ContaElimina -1 do
            begin
              for j := 0 to ListaEstados.ContaEstados -1 do
              begin
                if (Elimina[i].nome_estado=Estados[j].nome_estado) then begin
                    status:=j;
                end;
              end;
                ListaEstados.RemoveEstado(status);
            end;
 end;
```

Fonte: Serafin (2009, p.56).

Conforme Serafin (2009), os estados mortos foram eliminados através da procedure *Elimina_estados_mortos*, que consiste no estado que não é inicial ou final e não ser origem de nenhuma transição, apenas si próprio. O método *RemoveConexoes* da classe *Tconexao* efetua a eliminação, onde após é inserido em um vetor para posteriormente ser eliminado do AF (SERAFIN, 2009). A figura 28 apresenta o código da procedure implementada.

Figura 28 - Eliminação de estados mortos.

```
procedure Tfrm_minimizacao.Elimina_estados_mortos;

var
h, i, C, R, j, n, status, status1, status2 :integer;
S , T:string;
ListaElimina : TElimina;

begin
status1:=0;
status2:=0;

status:=0;
NElimina:=0;
for i:=0 to ListaEstados.ContaEstados-1 do
    begin
  for j := 0 to ListaConexoes.ContaConexoes -1     do
     begin
          if (Estados[i].nome_estado <> Conexao[j].estado_origem)and(Estados[i].estado_inicial=False)
          and (Estados[i].estado_final=False)then begin
            status1:=status1+1;
          end;

          if (Estados[i].nome_estado = Conexao[j].estado_destino)and(Estados[i].nome_estado = Conexao[j].estado_origem)
          and(Estados[i].estado_inicial=False)and (Estados[i].estado_final=False)then begin
            status1:=status1+1;
          end;
      end;

     if (status1=ListaConexoes.ContaConexoes)  then begin
         ListaConexoes.RemoveConexoes(Estados[i].nome_estado);
         ListaElimina.NovoElimina(Estados[i].nome_estado);
    end;
    status1:=0;
    status2:=0;
  end;
    status:=0;
         for i:=0 to ListaElimina.ContaElimina -1 do
             begin
               for j := 0 to ListaEstados.ContaEstados -1 do
               begin
                 if (Elimina[i].nome_estado=Estados[j].nome_estado)  then begin
                     status:=j;
                      end;
                 end;
                   ListaEstados.RemoveEstado(status);
                 end;
```

Fonte: Serafin (2009, p.57).

A terceira e ultima etapa de minimização é a eliminação dos estados equivalentes, onde foi implementado na procedure *Classes_Equivalencia.* A primeira parte desta etapa tem como objetivo identificar os estados finais e não finais do AFD, e armazena-los em um vetor de TClasse. Na figura 29 pode ser vista a declaração da classe TClasse, que tem os métodos para armazenar os estados finais e não

finais, e a figura 30 demonstra os estados sendo inseridos através dos métodos de TClasse (SERAFIN, 2009).

Figura 29 - Declaração Tclasse.

```
TClasse = class
nome_estado1 :string;


procedure Insere_ClasseFinais(Rotulo1:string);
function ContaClasseFinais: integer;
procedure Insere_ClasseNaoFinais(Rotulo1:string);
function ContaClasseNaoFinais: integer;
end;
```

Fonte: Serafin (2009, p.58).

Figura 30 - Inserção estados finais e não finais.

```
for i:=0 to ListaEstados.ContaEstados -1 do begin
      if (Estados[i].estado_final = True) then begin
        ListaClasseFinais.Insere_ClasseFinais(Estados[i].nome_estado);
      end;
      if (Estados[i].estado_final = False) then begin
        ListaClasseNaoFinais.Insere_ClasseNaoFinais(Estados[i].nome_estado);
      end;
end;
```

Fonte: Serafin (2009, p.58).

Após a identificação dos estados finais e não finais é feito um comparativo entre as transições com o intuito de buscar padrões, onde serão inseridos em um vetor de TClasse_equiv, com a possibilidade de posteriormente se tornarem estados únicos. Os algoritmos para estados finais e não finais são representados nas figuras 31 e 32, respectivamente.

Figura 31 - Identificação dos estados finais equivalentes.

```
for i:=0 to ListaClasseFinais.ContaClasseFinais-1  do begin
    for h:=i+1 to ListaClasseFinais.ContaClasseFinais-1  do begin
      if(ClasseFinais[i].nome_estado1<>ClasseFinais[h].nome_estado1)  then begin
        for j:=0 to ListaConexoes.ContaConexoes-1 do begin
          if (Conexao[j].estado_origem=ClasseFinais[i].nome_estado1)  then begin
              if (Conexao[j].estado_destino<>'') then begin
                for g:=0 to ListaEstados.ContaEstados -1 do begin
                  if (Conexao[j].estado_destino = Estados[g].nome_estado)  then begin
                    if (Estados[g].estado_final=True) then begin
                        status1[e] := 1;
                    end;
                    if (Estados[g].estado_final=False)then begin
                        status1[e] := 2;
                    end;
                    e:=e+1;
                  end;
                end;
              end;
          end;
        end;

        for k:=0 to ListaConexoes.ContaConexoes-1 do begin
          if (Conexao[k].estado_origem=ClasseFinais[h].nome_estado1)  then begin
            if (Conexao[k].estado_destino<>'') then begin
              for g:=0 to ListaEstados.ContaEstados -1 do begin
                if (Conexao[k].estado_destino = Estados[g].nome_estado)  then begin
                  if (Estados[g].estado_final=True)  then begin
                      status2[d] := 1;
                  end;
                  if (Estados[g].estado_final=False)then begin
                      status2[d] := 2;
                  end;
                  d:=d+1;
                end;
              end;
            end;
          end;
      end;
    end;

    indicador:=0;
   if(d=e) then begin
     for p:= 0 to  ListaAlfabeto.ContaAlfabeto -1 do begin
        if (status1[p] = status2[p]) then begin
            indicador:= indicador +1;
        end;
     end;
   end;
   if (indicador = ListaAlfabeto.ContaAlfabeto)  then begin
     ListaEquiv.Insere_ClasseEquivalencia(ClasseFinais[i].nome_estado1, ClasseFinais[h].nome_estado1,
     ClasseFinais[i].nome_estado1+ClasseFinais[h].nome_estado1);
     indice:=indice+1;
   end;
   d:=0;
   e:=0;

  end;
 end;
indice:=0;
end;
```

Fonte: Serafin (2009, p.59).

Figura 32 - Identificação dos estados não finais equivalentes.

```
for i:=0 to ListaClasseNaoFinais.ContaClasseNaoFinais-1  do begin
        for h:=i+1 to ListaClasseNaoFinais.ContaClasseNaoFinais-1  do begin
          if(ClasseNaoFinais[i].nome_estado1<>ClasseNaoFinais[h].nome_estado1) then begin
              for j:=0 to ListaConexoes.ContaConexoes-1 do begin
                  if (Conexao[j].estado_origem=ClasseNaoFinais[i].nome_estado1) then begin
                      if (Conexao[j].estado_destino<>'') then begin
                          for g:=0 to ListaEstados.ContaEstados -1 do begin
                              if (Conexao[j].estado_destino = Estados[g].nome_estado) then begin
                                  if (Estados[g].estado_final=True) then begin
                                      status1[e] := 1;
                                  end;
                                  if (Estados[g].estado_final=False)then begin
                                      status1[e] := 2;
                                  end;
                                   e:=e+1;
                              end;
                          end;
                      end;
                  end;
              end;

              for k:=0 to ListaConexoes.ContaConexoes-1 do begin
                  if (Conexao[k].estado_origem=ClasseNaoFinais[h].nome_estado1) then begin
                    if (Conexao[k].estado_destino<>'') then begin
                        for g:=0 to ListaEstados.ContaEstados -1 do begin
                            if (Conexao[k].estado_destino = Estados[g].nome_estado) then begin
                                if (Estados[g].estado_final=True) then begin
                                    status2[d] := 1;
                                end;
                                if (Estados[g].estado_final=False)then begin
                                    status2[d] := 2;
                                end;
                                d:=d+1;
                            end;
                        end;
                    end;
                end;
            end;

        indicador:=0;
        if(d=e) then begin
            for p:= 0 to  ListaAlfabeto.ContaAlfabeto -1 do begin
                if (status1[p] = status2[p]) then begin
                    indicador:= indicador +1;
                end;
            end;
        end;
        if (indicador = ListaAlfabeto.ContaAlfabeto) then begin
            ListaEquiv.Insere_ClasseEquivalencia(ClasseNaoFinais[i].nome_estado1,
            indice:=indice+1;
        end;
        d:=0;
        e:=0;
    end;
  end;
indice:=0;
end;
```

Fonte: Serafin (2009, p.60).

O último passo do processo de eliminação dos estados equivalentes, ilustrado na figura 33, é verificar a união das transições de cada par de estados

contido no vetor *TClasse_equiv,* com o objetivo de identificar se transitam com o mesmo símbolo de entrada para estados da mesma classe de equivalência. Caso positivo, suas transições são unidas e definidas como estado inicial ou final de acordo com o tipo dos estados. Já se der negativo, o par de estados é eliminado do vetor (SERAFIN, 2009).

Figura 33 - Identificação dos estados equivalentes.

```
      while (b<ListaEquiv.ContaClasseEquivalencia) do begin
          for i:= 0 to ListaConexoes.ContaConexoes-1 do begin
              if (Conexao[i].estado_origem = Equivalencia[b].nome_estado1) then begin
                  status3[n]:=Conexao[i].estado_destino;
                  n:=n+1;
              end;
              if (Conexao[i].estado_origem = Equivalencia[b].nome_estado2) then begin
                  status4[m]:=Conexao[i].estado_destino;
                  m:=m+1;
              end;
          end;
          for i:=0 to ListaAlfabeto.ContaAlfabeto-1 do begin
              status7:=status3[i]+status4[i];
                  if (status3[i]=status4[i]) then begin
                    status6:=status6+1;
                  end;
                  if (status3[i]<>status4[i]) then begin
                    for j:=0 to ListaEquiv.ContaClasseEquivalencia-1 do begin
                      if (status7=Equivalencia[j].estado_equiv) then begin
                          status6:=status6+1;
                      end;
                    end;
                  end;
            end;
           if (status6<ListaAlfabeto.ContaAlfabeto) then begin
               ListaEquiv.Exclui_ClasseEquivalencia(b);
               b:=b-1;
           end;
      n:=0;
      m:=0;
      status6:=0;
      status7:='';
      b:=b+1;

  end;
```

Fonte: Serafin (2009, p.61).

E para finalizar os algoritmos utilizados no AFLAB, Serafin (2009) implementou o módulo de geração de gramática regular a partir de um autômato finito determinístico, onde permite gerar gramática de AFD e de AFND. No código implementado (figura 34) foi utilizado às classes principais do AFLAB (*Testados, Talfabeto e Tconexao*), percorrendo-as na identificação de cada estado do AF. Cada transição é colocada no vetor *status3*, que verifica todas as transações possíveis do

estado e envia um somatório das informações armazenadas para vetor status2, ou seja, o símbolo de entrada e o estado destino de cada estado (SERAFIN, 2009).

Figura 34 - Gramática a partir de um AF.

```
for i:=0 to ListaEstados.ContaEstados-1 do begin
  for n:=0 to ListaAlfabeto.ContaAlfabeto -1 do begin
    for j:=0 to ListaConexoes.ContaConexoes-1 do begin

      if (Conexao[j].estado_origem=Estados[i].nome_estado)and(Conexao[j].Caractere=Alfabeto[n].letra_alfabeto) then begin

          if (h=0) then begin
               for a:=0 to ListaEstados.ContaEstados -1 do begin
                  if (Estados[a].nome_estado=Conexao[j].estado_destino) then begin
                    if (Estados[a].estado_final=True) then begin
                        status3[h]:=Conexao[j].Caractere+Conexao[j].estado_destino+'/'+Conexao[j].Caractere;
                        indicador1:=1;
                    end;
                    if (Estados[a].estado_final=False) then begin
                        status3[h]:=Conexao[j].Caractere+Conexao[j].estado_destino;
                    end;
                  end;
               end;
          end;

          if(h>0) then begin
                for a:=0 to ListaEstados.ContaEstados -1 do begin
                   if (Estados[a].nome_estado=Conexao[j].estado_destino) then begin
                     if (Estados[a].estado_final=True)then begin
                        if (indicador1=0) then begin
                        status3[h]:='/'+Conexao[j].Caractere+Conexao[j].estado_destino+'/'+Conexao[j].Caractere;
                        indicador1:=1;
                        end
                        else begin
                        status3[h]:='/'+Conexao[j].Caractere+Conexao[j].estado_destino;
                        end;
                     end;
                     if (Estados[a].estado_final=false) then begin
                        status3[h]:='/'+Conexao[j].Caractere+Conexao[j].estado_destino;
                     end;
                   end;
                end;
          end;
      h:=h+1;
      indicador:=1;
      end;
    end;
  indicador1:=0;
  end;
    if (indicador=0) then begin
      status2[p]:='i';
    end;

    for m:=0 to h-1 do begin
      status2[p]:=status2[p]+status3[m];
    end;
```

Fonte: Serafin (2009, p.63).

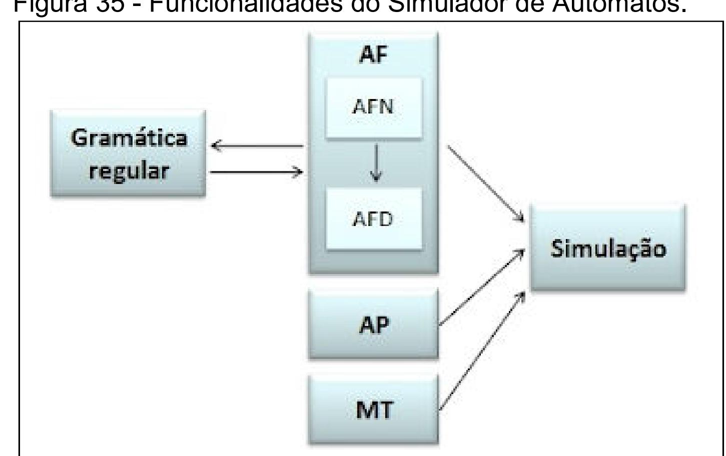
5 TRABALHOS CORRELATOS

Existem diversos projetos desenvolvidos por universidades que abrangem os estudos de linguagens formais e teoria da computação, onde alguns resultaram no desenvolvimento de ferramentas de auxilio para essas áreas. Neste capítulo serão apresentados alguns dos principais aplicativos semelhantes ao AFLAB encontrados durante as pesquisas realizadas:

a) sob a direção de Susan Rodger, diversos estudantes fazem parte do desenvolvimento do *Java Formal Language and Automata Package* (JFLAB), que teve início na instituição de ensino superior Rensselaer Polytechnic Institute por volta de 1990, e posteriormente mudou-se para Duke University em 1994 junto com seu diretor. O JFLAP é um software para experimentos de temas em linguagens formais, onde disponibiliza ferramentas gráficas para a simulação e criação de AFD, AFND, expressões regulares e gramáticas regulares, permitindo fazer a transformação de AFND para AFD, minimização de AFD, e as conversões de AFND para uma expressão ou gramática regular, e também de uma expressão ou gramática regular para um AFND. Essas são algumas de suas funcionalidades, onde disponibiliza também ambientes para manipulação de linguagens livres de contexto, Linguagens recursivamente enumeráveis e L-Systems (RODGER, 1990);
b) criado por Charbel Szymanski, e primeira versão publicada no final de 2005 como projeto de conclusão do curso de graduação em Ciência da Computação, o *Auger* é um ambiente de construção e simulação de autômatos finitos, com o objetivo de disponibilizar uma ferramenta de apoio didático no ensino de linguagens formais. Sua ultima versão 3.1 tem como principais funcionalidades criar autômatos finitos na forma gráfica a partir do diagrama de estados, executar algoritmos de manipulação de autômatos finitos, simular cadeias de entradas nos autômatos finitos, gerar autômatos finitos através de expressões regulares ou gramáticas regulares, gerar expressões regulares ou gramáticas regulares a partir de autômatos finitos (SZYMANSKI, 2005);

c) o *Simulador de Autômatos* é um aplicativo que permite a criação, simulação e conversão de modelos formais, criado com o objetivo de auxiliar o aprendizado de Linguagens Formais e Autômatos Finitos. Foi desenvolvido por Neilton Gonçalves Ribeiro Junior inicialmente como projeto de iniciação científica da disciplina de Teoria da Computação e posteriormente continuado como Trabalho de Conclusão de Curso de Engenharia da Computação da Universidade de Uberaba (RIBEIRO JUNIOR, 2009). A figura 35 abaixo ilustra suas funcionalidades:

Figura 35 - Funcionalidades do Simulador de Autômatos.



Fonte: Ribeiro Junior (2009).

d) implementado na linguagem Java durante o curso de pós-graduação da Universidade Federal de Minas Gerais, o *Language Emulator* é uma ferramenta de auxilio no aprendizado das teorias dos autômatos, onde permite a manipulação de expressões regulares, gramáticas regulares, AFD, AFND, AFND com transições lambda, máquinas de Moore e de Mealy, entre outras funcionalidades (Vieira; Vieira; Vieira, 2003).

e) Inspirado por uma turma da disciplina de Teoria da Computação da Universidade de São Francisco, Califórnia, foi desenvolvido em Java a ferramenta chamada *Visual Automata Simulator*, aplicação com interface gráfica que possibilita aos usuários a criação e simulação de autômatos finitos determinísticos ou não determinísticos, bem como também a transformação de não determinísticos para determinístico. Permite também a manipulação de máquinas de Turing (BOVET, 2004).

6 AF-GR WEB

Este capítulo apresenta o desenvolvimento da aplicação WEB proposta como objetivo deste trabalho. A aplicação disponibiliza um ambiente gráfico de fácil utilização, onde podem ser criados autômatos finitos na forma gráfica através de botões com finalidades especificas e ações com o mouse na tela. Há três módulos de criação diferentes na aplicação com suas respectivas funcionalidades, onde inicialmente temos o módulo de autômatos finitos, que pode ser testado uma sentença, transformar um AFND em AFD, fazer a minimização, gerar a gramática regular através do AF desenhado e desenhar um AF através da inclusão de uma gramática. Nos outros dois módulos podem ser simulados máquina de Mealy ou Moore, onde temos imagens como tipo de saída associadas às transições (Mealy) ou aos estados (Moore).

No desenvolvimento da aplicação foi utilizado o HTML5 na criação das páginas web, junto com a linguagem CSS para estilizar as páginas. Para facilitar o desenvolvimento foi utilizado o Bootstrap, que é um framework front-end que disponibiliza uma coleção de vários elementos e funções personalizáveis para projetos web. O ambiente gráfico foi desenvolvido no elemento canvas do HTML5 utilizando o framework KineticJS, que permite desenhar formas e imagens no canvas com mais facilidade. E por fim, foi utilizado o javascript para fazer as ações do usuário com os elementos da página.

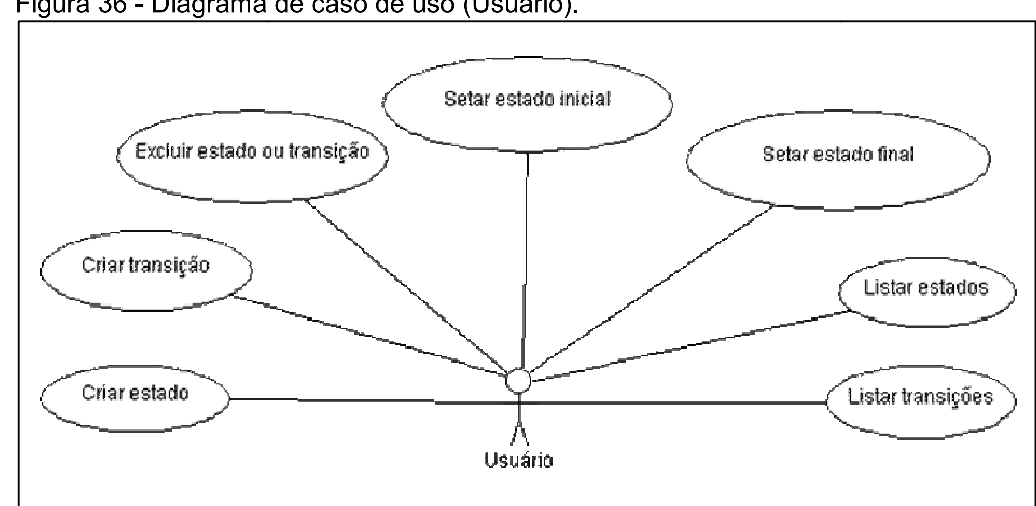
6.1 METODOLOGIA

Para atingir os objetivos propostos neste projeto de pesquisa foram necessárias algumas etapas metodológicas, onde primeiramente foi feito um levantamento bibliográfico, buscando um melhor conhecimento das linguagens formais, com o foco voltado para os autômatos finitos e as gramáticas regulares. Foi feito um estudo sobre os AFD e AFND, suas diferenças e a forma de transformação de AFND em AFD. O modo de minimizar um AF, autômatos finitos com saída (máquina de Mealy e Moore), como gerar um AF através de uma GR e uma GR através de um AF também estiveram neste estudo. Por fim, foi disponibilizado o código fonte do aplicativo AFLAB, que complementou esse estudo através da compreensão dos seus algoritmos já implementados.

Foi de grande importância também um pequeno estudo da utilização dos frameworks Bootstrap e KineticJS para utilizar no desenvolvimento da aplicação. O HTML5, CSS e javascript foram escolhidos por serem ferramentas mais utilizadas nos aplicativos WEB atualmente.

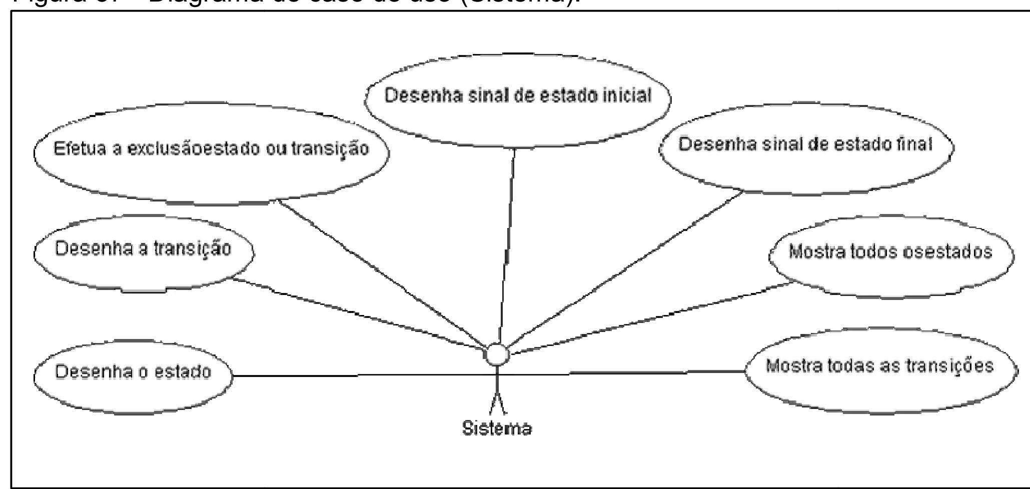
Após o levantamento bibliográfico foi dado início ao desenvolvimento do aplicativo WEB. Primeiramente foi feito a modelagem UML, onde foram obtidos os diagramas apresentados nas figuras 36 e 37.

Figura 36 - Diagrama de caso de uso (Usuário).



Fonte: Do autor.

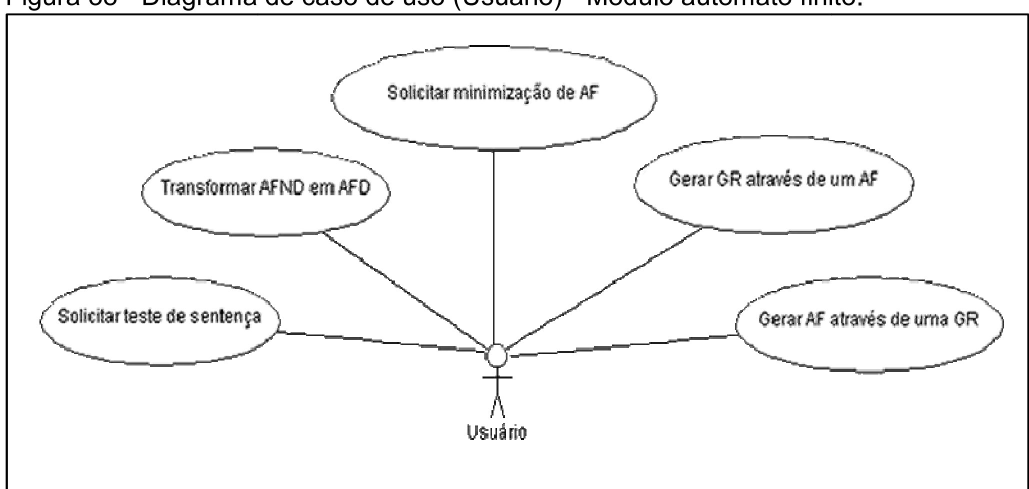
Figura 37 - Diagrama de caso de uso (Sistema).



Fonte: Do autor.

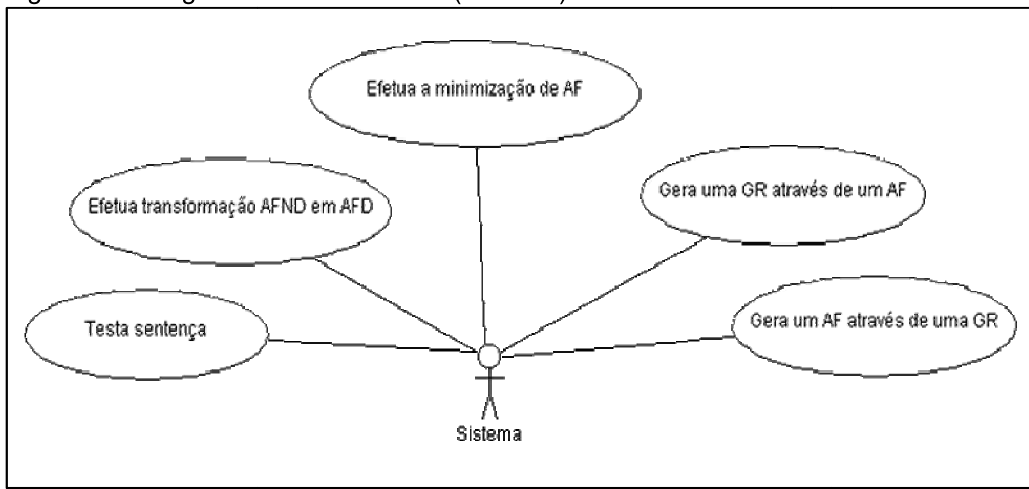
As figuras 36 e 37 representão os diagramas de caso de uso genéricos para todos os módulos, ou seja, são funcionalidades acessíveis em todos os módulos do aplicativo. As figuras 38, 39, 40, 41, 42 e 43 representam os diagramas das funcionalidades por módulo.

Figura 38 - Diagrama de caso de uso (Usuário) - Módulo autômato finito.



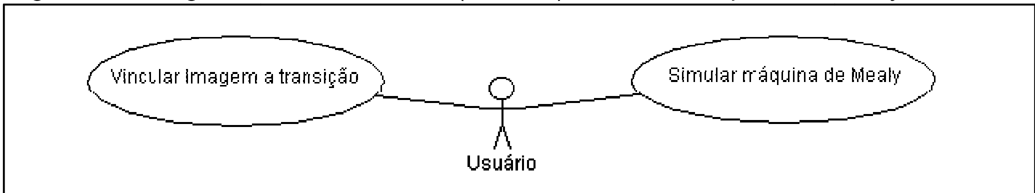
Fonte: Do autor.

Figura 39 - Diagrama de caso de uso (Sistema) - Módulo autômato finito.



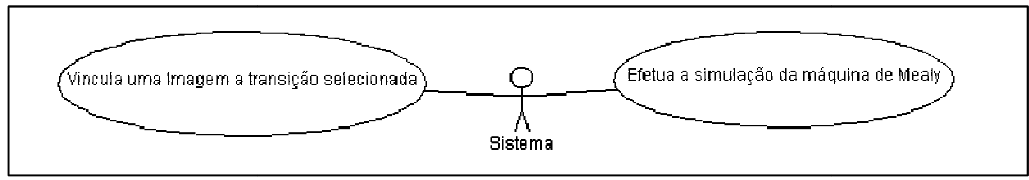
Fonte: Do autor.

Figura 40 - Diagrama de caso de uso (Usuário) - Módulo máquina de Mealy.



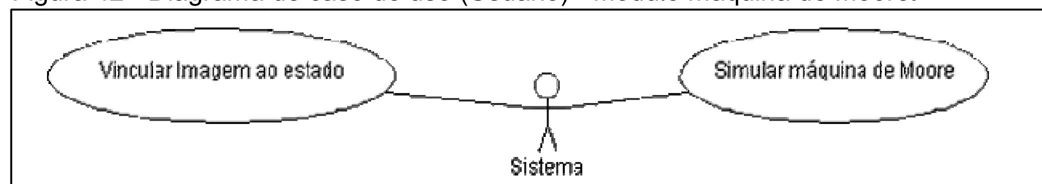
Fonte: Do autor.

Figura 41 - Diagrama de caso de uso (Sistema) - Módulo máquina de Mealy.



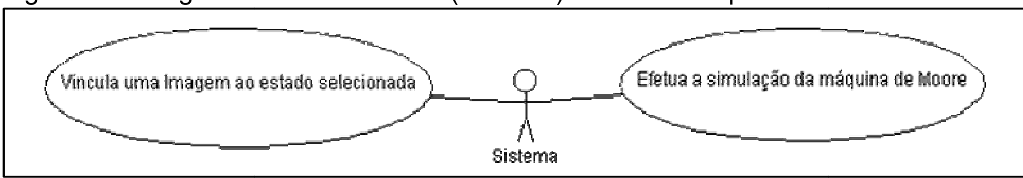
Fonte: Do autor.

Figura 42 - Diagrama de caso de uso (Usuário) - Módulo máquina de Moore.



Fonte: Do autor.

Figura 43 - Diagrama de caso de uso (Sistema) - Módulo máquina de Moore.



Fonte: Do autor.

6.2 DESENVOLVIMENTO

Como estrutura de armazenamento do aplicativo foi definida duas classes: Estado e Transição. A classe estado tem como atributos o nome do estado, se é inicial e final, o objeto com as informações necessárias para desenhar na tela, e a imagem que só será utilizada na máquina de Moore. Já na classe Transição os atributos são o estado origem, estado destino, o símbolo da transição e a imagem que será usada somente na máquina de Mealy. As classes são armazenadas em dois arrays dinâmicos, sendo um a lista de estados e outro a lista de transições. Na figura 45 e 46 temos as classes que são a base para os algoritmos implementados na aplicação.

Figura 44 - Classe Estado

```
function Estado(nome, obj) {
    this.nome = nome;
    this.eInicial = false;
    this.eFinal = false;
    this.objeto = obj;
    this.img = undefined;

    this.getImg = function () {
        return this.img;
    };

    this.setImg = function (img) {
        this.img = img;
    };

    this.getNome = function () {
        return this.nome;
    };

    this.setNome = function (n) {
        this.nome = n;
    };

    this.getInicial = function () {
        return this.inicial;
    };

    this.setInicial = function (i) {
        this.inicial = i;

    this.getFinal = function () {
        return this.eFinal;
    };

    this.setFinal = function (f) {
        this.eFinal = f;
    };

    this.getObjeto = function () {
        return this.objeto;
    };

    this.setObjeto = function (o) {
        this.objeto = o;
    };
}
```

Fonte: Do autor.

Figura 45 - Classe Transição

```
function Transicao(e1, e2, simbolo) {
    this.simbolo = simbolo;
    this.origem = e1;
    this.destino = e2;
    this.img = undefined;

    this.getImg = function () {
        return this.img;
    };

    this.setImg = function (img) {
        this.img = img;
    };

    this.getSimbolo = function () {
        return this.simbolo;
    };

    this.setSimbolo = function (s) {
        this.simbolo = s;
    };

    this.getOrigem = function () {
        return this.origem;
    };

    this.setOrigem = function (o) {
        this.origem = o;
    };

    this.getDestino = function () {
        return this.destino;
    };

    this.setDestino = function (d) {
        this.destino = d;
    };

}
```

Fonte: Do autor.

6.2.1 Módulo Autômato Finito

Neste módulo temos as principais funcionalidades desenvolvidas nessa aplicação, onde pode ser testada uma sentença, fazer a transformação de um AFND em AFD, minimizar o autômato finito, gerar GR através de um AF e um AF através de uma GR. Nos próximos capítulos são apresentados à implementação dessas funcionalidades.

6.2.1.1 Reconhecedor de sentenças

Para reconhecer uma sentença foi implementado a função testaSentenca, que recebe como parâmetro uma lista de transições, a sentença, o estado inicial e os estado finais do AF. A função faz uma verificação se o AF tem estado inicial e estado final, onde caso não tenha é retornando “undefined” e uma mensagem é mostrada para o usuário. Se passar da verificação o estado inicial é colocado como estado atual e é iniciado o reconhecimento, percorrendo a sentença da esquerda para direita, pegando cada símbolo e buscando a transição referente do estado atual com o símbolo. Caso tenha a transição, o estado destino é colocado como estado atual e é chamada recursivamente a função testaSentenca. Esse processo é feito até ler todos os símbolos da sentença. Caso a função testaSentenca retornar alguma vez “false” ou não houver uma transição para um símbolo, a sentença é dada como não reconhecida. Se retornar true e o estado atual for um estado final a sentença é dada como reconhecida, mas se não for final não é reconhecida novamente. Na figura 48 pode ser visto a implementação da função testaSentenca.

Figura 46 - Função testa sentença.

```
function testaSentenca(listaTransicoes, estadoInicial, estadosFinais, sentenca, i) {
  var c, simbolo, estadoAtual, transicaoEfetuada;
  var estadoAux = estadoInicial;

  if(estadoInicial != "" && estadosFinais[0] != undefined){

      while(sentenca.charAt(i) != ""){
          estadoAtual = estadoAux;
          simbolo = sentenca.charAt(i);
          transicaoEfetuada = false;
          for(c = 0; c < listaTransicoes.length; c++){
              if(estadoAtual ==  listaTransicoes[c].origem.getName() && simbolo == listaTransicoes[c].simbolo.getText()){

                  estadoAux = listaTransicoes[c].destino.getName();
                  transicaoEfetuada = true;
                  var valida = testaSentenca(listaTransicoes, estadoAux, estadosFinais, sentenca, i+1);
                  if(valida == true){
                      return true;
                  }
              }
          }
          if(transicaoEfetuada == false){
              return false;
          }
          i++;
      }
       estadoAtual = estadoAux;
       for(c = 0; c < estadosFinais.length; c++){
           if(estadoAtual == estadosFinais[c]){
            return true;
           }
       }
       return false;
    }else{
       return undefined
    }
}
```

Fonte: Do autor.

A função para reconhecer sentença serve para AFD e AFND, pois a recursividade nessa função permitiu esse reconhecimento para os dois tipos de AF.

6.2.1.2 Transformação de AFND em AFD

Para aplicar a transformação dever ser desenhado um AFND, ou seja, um AF que permitem a transição para mais de um estado com uma única combinação estado/símbolo de entrada. Para efetuar essa transformação foi criado a função afndToAfd. Está função recebe como parâmetro a lista de estados, lista de transições e os símbolos do AFND desenhado.

Para iniciar a transformação o AFND desenhado deve conter um estado inicial e pelo menos um estado final. É verificado também se o AF desenhado é um AFND, caso não seja a transformação não é executada pois não é necessário. Passando por esse requisitos, é incluído um estado novo chamado vazio, e criado uma tabela de transição do AFND, onde a variável newTransicoes recebe uma lista de transições unindo as não determinísticas. Após é percorrida todas as transições e onde houver não determinismo é incrementado um novo estado em newTransicoes

através de sua união, onde o estado será final se algum dos estados unido for final. Já as transições desse novo estado são definidas através da união das transições de cada estado presentes na união. Esse processo é feito até que não tenha mais nenhuma ocorrência de não determinismo dentro de newTransicoes. Terminado o processo é chamada a função desenhaAFD, onde é feito a limpeza da tela e desenhado o AFD. Por fim, o estado vazio incluído no início da função é excluído. As figuras 47 e 48 apresentam a implentação desta transformação.

Figura 47 - Transformação de AFND em AFD.

```
function afndToAfd(listaEstados, listaTransicoes, simbolos) {

    var afnd = isAFND(simbolos);
    var i = 0, c = 0, z = 0, x = 0, p = 0, j = 0, y = 0;
    var estados = [], newTransicoes = [], u, u2, transicao, estado, encontrou;

    if(afnd == true){
        u = new Estado("&", Ponto(50, 50));
        listaEstados.unshift(u);
        while(listaEstados[i] != undefined){
            c = 0;
            for(c; c < simbolos.length; c++){
                z = 0;
                u = "";
                for(z; z < listaTransicoes.length; z++){
                    if(listaTransicoes[z].origem.getName() == listaEstados[i].nome &&
                        listaTransicoes[z].simbolo.getText() == simbolos[c]){

                        if(u == ""){
                            u = listaTransicoes[z].destino.getName();
                        }else{
                            u = u + "," + listaTransicoes[z].destino.getName();
                        }
                    }
                }

                u2 = u.split(",");
                u = removeDuplicatedArray(u2, false);
                u.sort();
                u = u.join(",");
                if(u != ""){
                    transicao = new Transicao(listaEstados[i].nome, u, simbolos[c]);
                    newTransicoes.push(transicao);
                }else{
                    transicao = new Transicao(listaEstados[i].nome, "&", simbolos[c]);
                    newTransicoes.push(transicao);
                }
            }
            i++;
        }

        for(i = 0; i < newTransicoes.length; i++){
            estados = newTransicoes[i].destino.split(",");
            if(estados.length >= 2){
                encontrou = false;
                for(x = 0; x < listaEstados.length; x++){
                    if(listaEstados[x].nome == newTransicoes[i].destino){
                        encontrou = true;
                    }
                }
```

Fonte: Do autor.

Figura 48 - Continuação transformação de AFND em AFD.

```
            if(encontrou != true){
                x = Math.floor((100) + Math.random() * (500 - 100));
                y = Math.floor((80) + Math.random() * (220 - 80));
                p = new Ponto(x, y);
                estado = new Estado(newTransicoes[i].destino, p);
                for(x = 0; x < estados.length; x++){
                    for(c = 0; c < listaEstados.length; c++){
                        if(listaEstados[c].nome == estados[x] && listaEstados[c].eFinal == true){
                            estado.eFinal = true;
                        }
                    }
                }
                listaEstados.push(estado);
                for(c = 0; c < simbolos.length; c++){
                    u = "";
                    for(x = 0; x < estados.length; x++){
                        for(p = 0; p < newTransicoes.length; p++){
                            if(newTransicoes[p].origem == estados[x] && simbolos[c]
                            == newTransicoes[p].simbolo){

                            if(newTransicoes[p].destino == "&"){
                                u = u + "";
                            }else{
                                 if(u == ""){
                                     u = newTransicoes[p].destino;
                                 }else{
                                     u = u + "," + newTransicoes[p].destino;
                                 }
                              }
                          }
                      }
                  }

                  u2 = u.split(",");
                  u = removeDuplicatedArray(u2, false);
                  u.sort();
                  u = u.join(",");

                  if(u != ""){
                      transicao = new Transicao(estado.nome, u, simbolos[c]);
                      newTransicoes.push(transicao);
                  }else{
                      transicao = new Transicao(estado.nome, "&", simbolos[c]);
                      newTransicoes.push(transicao);
                  }
              }
          }
      }
  }

  for(i = 1; i < listaEstados.length; i++){
      //alert("E:" + listaEstados[i].nome);
      for(x = 0; x < newTransicoes.length; x++){
          if(newTransicoes[x].origem == listaEstados[i].nome){
              newTransicoes[x].origem = alfabeto[i -1];
          }if(newTransicoes[x].destino == listaEstados[i].nome){
              newTransicoes[x].destino = alfabeto[i -1];
          }
      }
      listaEstados[i].nome = alfabeto[i -1];
  }

  desenhaAFD(listaEstados, newTransicoes);

  for(i = 0; i < listaEstados.length; i++){
      if(listaEstados[i].nome == "&"){
          excluirEstado = true;
          excluirEstados(listaEstados[i].nome);
          excluirEstado = false;
        }
     }
 }else{
     alert("O AF já é determinístico");
 }

}
```

Fonte: Do autor.

6.2.1.3 Minimização de Autômatos Finitos

A minimização consiste em deixar o AF com o menor numero de estados possíveis, através da eliminação dos estados inúteis, inacessíveis e equivalentes. Para isso ser feito foi implementado a função minimização, que recebe como parâmetro a lista de estado, lista de transições e símbolos do AF a ser minimizado. Para efetuar a transformação o AFND deve ter um estado inicial e pelo menos um estado final.

Antes de minimizar, o AF desenhado deve conter alguns pré-requisitos. Primeiro deve ser determinístico, então é chamada a função que transforma AFND em AFD, garantindo assim que ele será determinístico. Após é eliminado todos os estados inacessíveis através de um método desenvolvido com essa finalidade (figura 49). Dando continuidade é chamado a função defineFuncaoTotal (figura 50), que serve para tornar, se necessário, a função de transição em total, ou seja, um estado deve ter transições para todos os símbolos do AF. Depois de todos os pré-requisitos alcançados é chamada a função de minimização, criando inicialmente uma tabela que contém o relacionamento dos estados diferentes do AF (figura 51). Nesta tabela são marcados os estados trivialmente não equivalentes (figura 52), e estados não equivalentes conforme a função apresentada na figura 53. Terminando essas etapas o algoritmo de minimização unifica os estados equivalentes, onde a unificação é feita transitivamente, se os pares de estado equivalentes forem não finais, a unificação é feita em um único estado não final. O mesmo serve para os estado finais, unificando-os em um único estado final. E por ultimo se algum estado equivalente for inicial, então o estado unificado é inicial também.

Figura 49 - Função eliminar estados inacessíveis.

```
function eliminaEstadosInacessiveis(listaEstados, listaTransicoes){
    var acessiveis = [], estadosExcluir = [], add, isAces, i = 0, z = 0, a = 1, c = 0;
    var ei = retornaEstadoInicial(listaEstados);
    if(ei != false){
        acessiveis[0] = ei.nome;
    }
    while(acessiveis[z] != undefined){
        i = 0;
        for(i; i < listaTransicoes.length;i++){
            add = false;
            if(listaTransicoes[i].origem.getName() == acessiveis[z]){
                c = 0;
                for(c; c < acessiveis.length; c++){
                    if(acessiveis[c] == listaTransicoes[i].destino.getName()){
                        add = true;
                    }
                }
                if(add != true){
                    acessiveis.push(listaTransicoes[i].destino.getName());
                }
            }
        }
        z++;
    }
    for(c = 0; c < listaEstados.length; c++){
        isAces = false;
        for(i = 0; i < acessiveis.length; i++){
            if(listaEstados[c].nome == acessiveis[i]){
                isAces = true;
            }
        }
        if(isAces != true){
            estadosExcluir.push(listaEstados[c].nome);
        }

    }
    for(c = 0; c < estadosExcluir.length; c++){
        excluirEstado = true;
        excluirEstados(estadosExcluir[c]);
        excluirEstado = false;
    }
}
```

Fonte: Do autor.

Figura 50 - Método que define função total se necessário.

```
function defineFuncaoTotal(listaEstados, listaTransicoes, simbolos){
  //----- inclusão do estado D se necessario -----
  var z = 0, tem = false, dCriado = false;
  var i = 0, c, x;
  while(simbolos[z] != undefined){
      for(i = 0; i < listaEstados.length; i++){
          tem = false;
          for(c = 0; c < listaTransicoes.length; c++){
              if(listaEstados[i].nome == listaTransicoes[c].origem.getName() &&
                  listaTransicoes[c].simbolo.getText() == simbolos[z]){

                      tem = true;
              }
          }
          if(tem != true){
              if(dCriado == false){
                  var px = Math.floor((100) + Math.random() * (500 - 100));
                  var py = Math.floor((80) + Math.random() * (220 - 80));
                  var p = new Ponto(px, py);
                  desenhaEstado("d", p);
                  dCriado = true;
              }
              var ed;
              for(x = 0; x < listaEstados.length; x++){
                  if(listaEstados[x].nome == "d"){
                      ed = listaEstados[x];
                  }
              }
              if(listaEstados[i].nome == ed.nome){
                  desenhaTrasicaoMesmoEstado(listaEstados[i].objeto, simbolos[z]);
              }else{
                  desenhaTransicao(listaEstados[i].objeto, ed.objeto, simbolos[z]);
              }
          }
      }
      z++;
  }
}
```

Fonte: Do autor.

Figura 51 - Tabela com o relacionamento dos estados diferentes.

```
function minimizacao(listaEstados, listaTransicoes, simbolos){
    //tabela relacionando o estados distintos
    var tabela = [];
    var listaAnalise = [];
    var i = 0, c = 0, x = 0, z = 0, j = 0, k = 0;
    var t,t1,t2, pu, pv;
    for(i = 0; i < listaEstados.length; i++){
        c = i + 1;
        for(c; c < listaEstados.length; c++){
            t = new Transicao(listaEstados[i], listaEstados[c], "");
            tabela[x] = t;
            x++;
        }
    }
```

Fonte: Do autor.

Figura 52 - Marcar estados trivialmente não equivalentes na tabela.

```
for(i = 0; i < tabela.length; i++){
    if(tabela[i].origem.eFinal != tabela[i].destino.eFinal){
        tabela[i].simbolo = "X";
    }
}
```

Fonte: Do autor.

Figura 53 - Marcar estados não equivalentes na tabela.

```
    for(i = 0; i < tabela.length; i++){
        if(tabela[i].simbolo != "X"){
            for(c = 0; c < simbolos.length; c++){
                for(z = 0; z < listaTransicoes.length; z++){
                    if(tabela[i].origem.nome == listaTransicoes[z].origem.getName() &&
                        listaTransicoes[z].simbolo.getText() == simbolos[c]){

                        pu = listaTransicoes[z].destino.getName();
                    }if(tabela[i].destino.nome == listaTransicoes[z].origem.getName() &&
                        listaTransicoes[z].simbolo.getText() == simbolos[c]){

                        pv = listaTransicoes[z].destino.getName();
                    }
                }
                if(pu == pv){
                }if(pu != pv){
                    for(x = 0; x < tabela.length; x++){
                        if((tabela[x].origem.nome == pu && tabela[x].destino.nome == pv && tabela[x].simbolo != "X") ||
                            (tabela[x].origem.nome == pv && tabela[x].destino.nome == pu && tabela[x].simbolo != "X")){

                            t1 = new Transicao(pu, pv, "");
                            t2 = new Transicao(tabela[i].origem.nome, tabela[i].destino.nome, "");
                            t = new Transicao(t1, t2, "");
                            listaAnalise.push(t);

                        }
                       if((tabela[x].origem.nome == pu && tabela[x].destino.nome == pv && tabela[x].simbolo == "X") |
                           (tabela[x].origem.nome == pv && tabela[x].destino.nome == pu && tabela[x].simbolo == "X"))

                           tabela[i].simbolo = "x";
                           for(j = 0; j < listaAnalise.length; j++){
                               if(listaAnalise[j].origem.origem == tabela[i].origem.nome &&
                                   listaAnalise[j].origem.destino == tabela[i].destino.nome ||
                                     listaAnalise[j].origem.destino == tabela[i].origem.nome &&
                                         listaAnalise[j].origem.origem == tabela[i].destino.nome){
                                   for(k = 0; k < tabela.length; k++){
                                       if(tabela[k].origem.nome == listaAnalise[j].destino.origem &&
                                           tabela[k].destino.nome == listaAnalise[j].destino.destino ||
                                             tabela[k].origem.nome == listaAnalise[j].destino.destino &&
                                                 tabela[k].destino.nome == listaAnalise[j].destino.origem){

                                           tabela[k].simbolo = "x";
                                       }
                                   }

                               }
                           }

                       }
                   }
               }

           }
       }
   }

   unificarEstadosEquivalentes(listaEstados, listaTransicoes, simbolos, tabela);
}
```

Fonte: Do autor.

6.2.1.4 Gerar gramática regular a partir de um autômato finito.

Nesta funcionalidade do módulo autômato finito o usuário pode gerar a gramática regular referente ao AF desenhado. Foi implementado então a função automatoFinitoToGramatica, que recebe como parâmetro a lista de estados e de transições.

Para começar a gerar a gramática o sistema faz inicialmente uma verificação, onde caso o AF desenhado não possuir um estado inicial e pelo menos um estado final é apresentado uma mensagem alertando o usuário. Caso os

requisitos sejam atendido à função é chamada onde faz uma limpeza na tabela em que a gramática é exibida para o usuário. Em seguida são percorridas todas as transições referentes a cada estado, concatenando as produções em uma variável, e posteriormente exibido na tabela para o usuário, como mostra a figura 54.

Figura 54 - Função que gera GR a partir de AF.

```
function automatoFinitoToGramatica(listaEstados, listaTransicoes){
    var i = 0, z = 0, x = 0, num, producoes;
    var numOfRows, newRow, c0, c1, c2;
    var table = document.getElementById("tabelaAfToGr");
    while (table.rows.length > 0){
        table.deleteRow(0);
    }
    for(i = 0; i < listaEstados.length; i++){
        producoes = "";
        for(z = 0; z < listaTransicoes.length; z++){
            if(listaEstados[i].nome == listaTransicoes[z].origem.getName()){
                for(x = 0; x < listaEstados.length; x++){
                    if(listaEstados[x].nome == listaTransicoes[z].destino.getName()){
                        num = x;
                    }
                }
                if(producoes == ""){
                    producoes = listaTransicoes[z].simbolo.getText() +""+ alfabeto[num];
                }else{
                    producoes = producoes + "|" + listaTransicoes[z].simbolo.getText() +""+ alfabeto[num];
                }
            }
        }
        if(listaEstados[i].eFinal == true){
            if(producoes == ""){
                producoes = "ε";
            }else{
                producoes = producoes + "|" + "ε";
            }
        }
        table = document.getElementById("tabelaAfToGr");
        numOfRows = table.rows.length;
        newRow = table.insertRow(numOfRows);
        newRow.className = "info";

        c0 = newRow.insertCell(0);
        c0.innerHTML = "<input id='ntt"+ numOfRows +"' class='input-mini \n\
                    uneditable-input' type='text' style='text-align: right' \n\
                      value='"+ alfabeto[i] +"' disabled/>";
        c0.style.textAlign = "right";
        c1 = newRow.insertCell(1);
        c1.innerHTML = "<h4>→<h4/>";
        c1.style.textAlign = "center";
        c2 = newRow.insertCell(2);
        c2.innerHTML = "<input id='tt"+ numOfRows +"' class='input-large uneditable-input'\n\
                     type='text' value='"+ producoes +"' disabled/>";
        c2.style.textAlign = "left";

    }
}
```

Fonte: Do autor.

6.2.2.4 Gerar autômato finito a partir de uma gramática regular.

Este capítulo apresenta à funcionalidade inversa a apresentada no capítulo anterior, ou seja, agora o usuário pode gerar um AF através de uma GR

(figura 55). A função gramaticaToautomatoFinito ficou encarregada de fazer essa transformação.

A função começa limpando a tela de desenho. Após, percorre todas as linhas da gramática pegando o valor do primeiro campo e desenhando-os como um estado. Terminado essa etapa é incluído um novo estado chamado “qf” que será o estado final do AF, e depois percorrerá novamente as linhas da gramática informada, mas agora buscando os valores do segundo campo. Cada linha tem um campo com as produções referentes ao estado da mesma linha, logo as produções contidas no campo são separadas pelo caracter “|”, e desenhada uma transição para cada produção.

Figura 55 - Função que gera AF a partir de GR.

```
function gramaticaToautomatoFinito(){
    var i = 0, z = 0, x = 0, y, p, j = 0;
    var value, producoes, producao, transicao, obj, lTransicoes = [];
    var tabela = document.getElementById("tabelaGrToAf");
    var numOfRows = tabela.rows.length;

    limparTela();

    for(i = 0; i < numOfRows; i++){
        x = Math.floor((100) + Math.random() * (500 - 100));
        y = Math.floor((80) + Math.random() * (220 - 80));
        p = new Ponto(x, y);
        desenhaEstado(document.getElementById("nt" + i).value, p);
        if(i == 0){
            listaEstados[i].eInicial = true;
            desenhaSinalDeInicial(listaEstados[i].objeto);
        }
    }

    x = Math.floor((100) + Math.random() * (500 - 100));
    y = Math.floor((80) + Math.random() * (220 - 80));
    p = new Ponto(x, y);
    desenhaEstado("qf", p);
    desenhaSinalDeFinal(listaEstados[listaEstados.length -1].objeto);
    listaEstados[listaEstados.length -1].eFinal = true;

    for(i = 0; i < numOfRows; i++){
        value = document.getElementById("t" + i).value;
        producoes = value.split("|");
        for(z = 0; z < producoes.length; z++){
            producao = Trim(producoes[z]);
            if(producao.charAt(0) != "" && producao.charAt(1) != ""){
                for(j = 0; j < listaEstados.length; j++){
                    if(producao.charAt(1) == listaEstados[j].nome){
                        obj = listaEstados[j];
                    }
                }
                transicao = new Transicao(listaEstados[i], obj, producao.charAt(0));
                lTransicoes.push(transicao);
            }if(producao.charAt(1) == ""){
                transicao = new Transicao(listaEstados[i],
                        listaEstados[listaEstados.length -1], producao.charAt(0));
                lTransicoes.push(transicao);
            }
        }
    }

    for(i = 0; i < lTransicoes.length; i++){
        if(lTransicoes[i].origem.nome == lTransicoes[i].destino.nome){
            desenhaTrasicaoMesmoEstado(lTransicoes[i].origem.objeto, lTransicoes[i].simbolo);
        }else{
            desenhaTransicao(lTransicoes[i].origem.objeto,
                    lTransicoes[i].destino.objeto, lTransicoes[i].simbolo);
        }
    }
    transicoes.draw();
    layer.draw();
    estados.draw();

}
```

Fonte: Do autor.

6.2.2 Módulo Máquina de Mealy e Moore

Este módulo é responsável pela simulação das máquinas de Mealy e Moore, onde o usuário pode associar uma imagem a um estado (Moore) ou transição (Mealy) e verificar a saída de acordo à sentença inserida.

6.2.2.1 Simular máquina de Mealy

Para simular a MMe é necessário desenhar o AF e associar uma imagem para todas as transições existentes. Caso alguma transição fique sem imagem associada uma mensagem é exibida informando a transição. Também é necessário definir um estado inicial.

A função simularMaquinaDeMealy faz a simulação da máquina de Mealy, onde pega a sentença informada pelo usuário e percorre o AF como quando é feito o teste de sentença, mas agora a cada transição feita o array sequenciaTransicoes recebe a transição efetuada, até que a sentença termine. Depois é percorrido o array sequenciaTransicoes e exibido para o usuário em sequência as imagens associadas a cada transição contida no array. A figura 56 apresenta está função.

Figura 56 - Função que simula a MMe.

```
function simularMaquinaDeMealy(){
    var c, z = 0, i = 0, p, simbolo, next;
    var sequenciaEstados = [], sequenciaTransicoes = [];
    var sentenca = document.getElementById('sentenca2').value;
    var estadoInicial = undefined, transSemImg = undefined;

    for(z = 0; z < listaEstados.length; z++){
        if(listaEstados[z].eInicial == true){
            estadoInicial = listaEstados[z];
        }
    }
    for(z = 0; z < listaTransicoes.length; z++){
        if(listaTransicoes[z].img == undefined){
            if(transSemImg == undefined){
                transSemImg = "δ(" + listaTransicoes[z].origem.getName() + ", "
                    + listaTransicoes[z].simbolo.getText() + ") = "
                    + listaTransicoes[z].destino.getName() +"\n";
            }else{
                transSemImg = transSemImg + "δ(" + listaTransicoes[z].origem.getName()
                    + ", " + listaTransicoes[z].simbolo.getText() + ") = "
                    + listaTransicoes[z].destino.getName() +"\n";
            }
        }
    }
    z = 0;
    i = 0;
    if(estadoInicial != undefined){
        if(transSemImg == undefined){
            sequenciaEstados.push(estadoInicial);
            while(sequenciaEstados[i] != undefined){
                next = false;
                if(sentenca.charAt(z) != ""){
                    simbolo = sentenca.charAt(z);
                    for(c = 0; c < listaTransicoes.length; c++){
                        if(sequenciaEstados[i].nome ==  listaTransicoes[c].origem.getName()
                            && simbolo == listaTransicoes[c].simbolo.getText()){
                            sequenciaTransicoes.push(listaTransicoes[c]);
                            for(p = 0; p < listaEstados.length; p++){
                                if(listaEstados[p].nome == listaTransicoes[c].destino.getName()){
                                    sequenciaEstados.push(listaEstados[p]);
                                }
                            }
                            next = true;
                            z++;
                        }
                    }
                    if(next == false){
                        //alert("não tem transição para esse simbolo");
                    }
                }else{
                    //alert("fim sentença");
                }
                i++;
            }

            document.getElementById('list').innerHTML = "";

            for(i = 0; i < sequenciaTransicoes.length; i++){
                var img = localStorage.getItem(sequenciaTransicoes[i].simbolo.getText());
                var span = document.createElement('span');

                span.innerHTML = ['<img class="thumb img-polaroid" src="', img,'" title="Estado Atual:\n\
                                 ', imgs[i].destino.getName(), '\nEstado Anterior: \n\
                                    ', imgs[i].origem.getName(),'\nSimbolo: \n\
                                        ', imgs[i].simbolo.getText(),'\nSentença: \n\
                                            ', sentenca.substr(0,i+1),'"/>'].join('');

                document.getElementById('list').insertBefore(span, null);
            }
        }else{
            alert("Não foi selecionado imagem de saída na(s) transição(ões): "+ transSemImg);
        }
    }else{
        alert("Automato sem Estado inicial ou final");
    }
}
```

Fonte: Do autor.

6.2.2.1 Simular máquina de Moore

Para simular a MMo seguimos a mesma lógica apresentada para a simulação da MMe. Mas agora as imagens são associadas aos estados em vez de transições. Com isso, a cada transição feita o array sequenciaEstados recebe o estado destino, tendo assim armazenado a sequência de estados percorridos pela sentença informada pelo usuário. Depois é percorrido o array sequenciaEstados e exibido para o usuário em sequência as imagens associadas a cada estado contido no array. A figura 57 mostra a implementação desta função.

Figura 57 - Função que simula a MMo.

```
function simularMaquinaDeMoore(){
    var c, z = 0, i = 0, p, simbolo, next, sequenciaEstados = [];
    var sentenca = document.getElementById('sentenca2').value;
    var estadoInicial = undefined, estadosSemImg = undefined;

    for(z = 0; z < listaEstados.length; z++){
        if(listaEstados[z].eInicial == true){
            estadoInicial = listaEstados[z];
        }
        if(listaEstados[z].img == undefined){
            if(estadosSemImg == undefined){
                estadosSemImg = listaEstados[z].nome;
            }else{
                estadosSemImg = estadosSemImg + ", " +listaEstados[z].nome;
            }
        }
    }
    i = 0;
    z = 0;
    if(estadoInicial != undefined){
        if(estadosSemImg == undefined){
            sequenciaEstados.push(estadoInicial);
            while(sequenciaEstados[i] != undefined){
                next = false;
                if(sentenca.charAt(z) != ""){
                    simbolo = sentenca.charAt(z);
                   for(c = 0; c < listaTransicoes.length; c++){
                       if(sequenciaEstados[i].nome ==  listaTransicoes[c].origem.getName()
                               && simbolo == listaTransicoes[c].simbolo.getText()){
                           for(p = 0; p < listaEstados.length; p++){
                               if(listaEstados[p].nome == listaTransicoes[c].destino.getName()){
                                   sequenciaEstados.push(listaEstados[p]);
                               }
                           }
                           next = true;
                           z++;
                       }
                   }
                   if(next == false){
                   }
                }
                i++;
            }

            document.getElementById('list').innerHTML = "";

            for(c = 0; c < sequenciaEstados.length; c++){
                var img = localStorage.getItem(sequenciaEstados[c].nome);
                var span = document.createElement('span');
                if(c == 0){
                    span.innerHTML = ['<img class="thumb img-polaroid" src="', img,'"\n\
                                      title="Estado: ', sequenciaEstados[c].nome, '\nSentença: - \n\
                 }else{
                     span.innerHTML = ['<img class="thumb img-polaroid" src="', img,'"\n\
                                  title="Estado: ', sequenciaEstados[c].nome, '\nSentença: \n\
                                   ', sentenca.substr(0,c),'"/>'].join('');
                 }

                 document.getElementById('list').insertBefore(span, null);
             }
        }else{
            //alert("Não foi selecionado imagem de saída no(s) estado(s): "+ estadosSemImg);
        }
    }else{
        //alert("Escolha um ESTADO INICIAL...");
    }
}
```

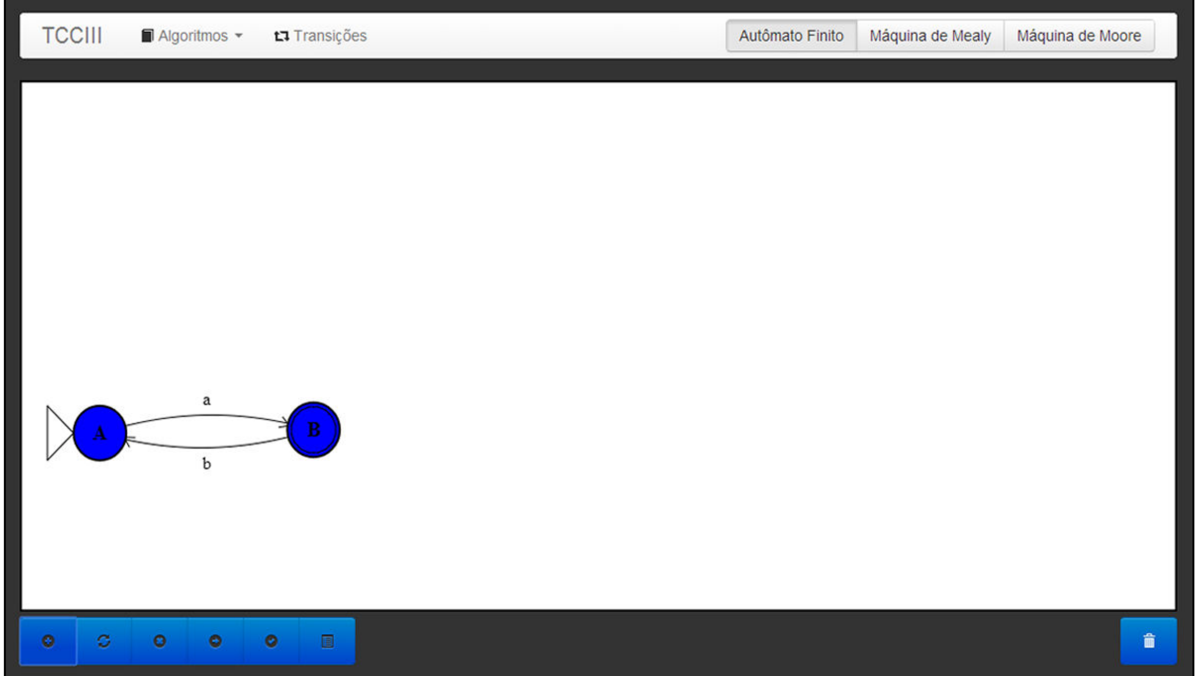
Fonte: Do autor.

6.3 RESULTADOS OBTIDOS

A aplicação disponibiliza um ambiente gráfico para a criação dos autômatos finitos, onde o usuário poderá introduzir o AF que ira tratar. Inicialmente é necessário selecionar no canto superior direito da aplicação um dos três tipos de AF a ser desenhado. O tipo autômato finito já vem por padrão e se for desenhar uma maquina de Mealy ou Moore, o tipo específico deve ser selecionado.

Após selecionar o tipo, o AF já pode ser desenhado através das opções de criação logo abaixo da tela de desenho, no canto inferior esquerdo. As opções são criar estado, criar transição, excluir, definir estado como estado inicial, definir estados finais e listar estados. Para criar um estado deve ser selecionada a opção referente à criação de estado e após clicar na tela de desenho, onde cada clique desenha um estado. Já para criar uma transição segue a mesma lógica anteriormente, mas agora deve clicar e segurar o botão esquerdo do mouse em cima do estado que vai ser origem da transição e após arrastar até o estado destino e largar o botão, mostrando assim à opção de inserir um símbolo para a transição. Nas opções de definir um estado inicial ou estado finais basta clicar em cima do estado desejado. Por fim tem a opção de listar os estados do AF onde mostra o nome, se é estado inicial e final. A figura 58 apresenta o ambiente gráfico.

Figura 58 - Ambiente gráfico



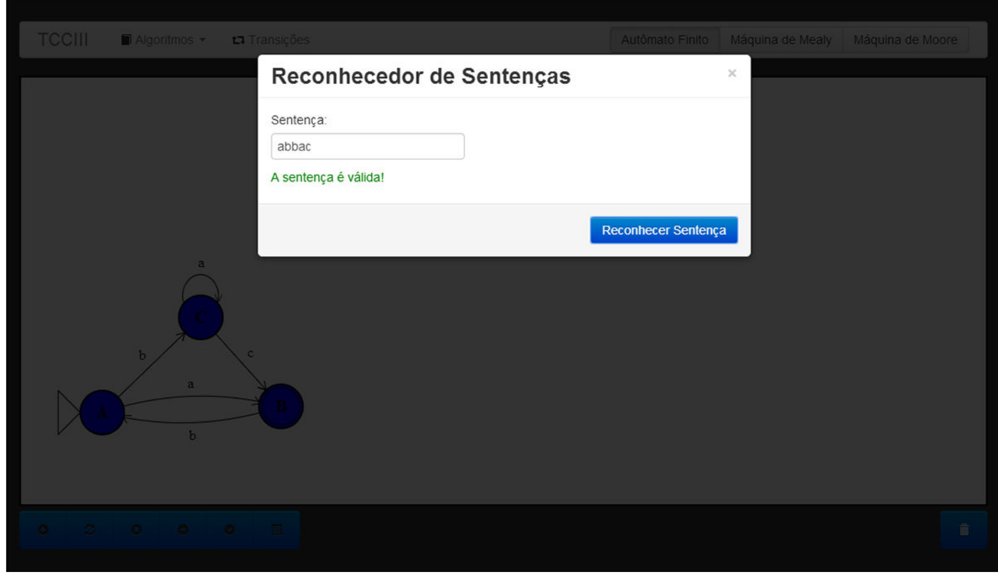
Fonte: Do autor.

No cabeçalho da página tem a opção algoritmo (figura 59), que como o nome já diz, disponibiliza os algoritmos referentes ao tipo selecionado, ou seja, se tiver selecionado a máquina de Moore, aparecerá somente os algoritmos aplicados a esta opção. Seguindo no cabeçalho tem a opção transições, onde lista todas as transições criadas. Se tiver selecionado para desenhar maquina de Mealy, a lista de transições terá mais uma coluna onde será utilizada para associar saída para as transições. Por fim tem o botão que limpa toda tela, localizado no canto inferior direito.

Figura 59 - Tela opções de algoritmos.

Fonte: Do autor.

Figura 60 - Tela reconhecedor de sentença.

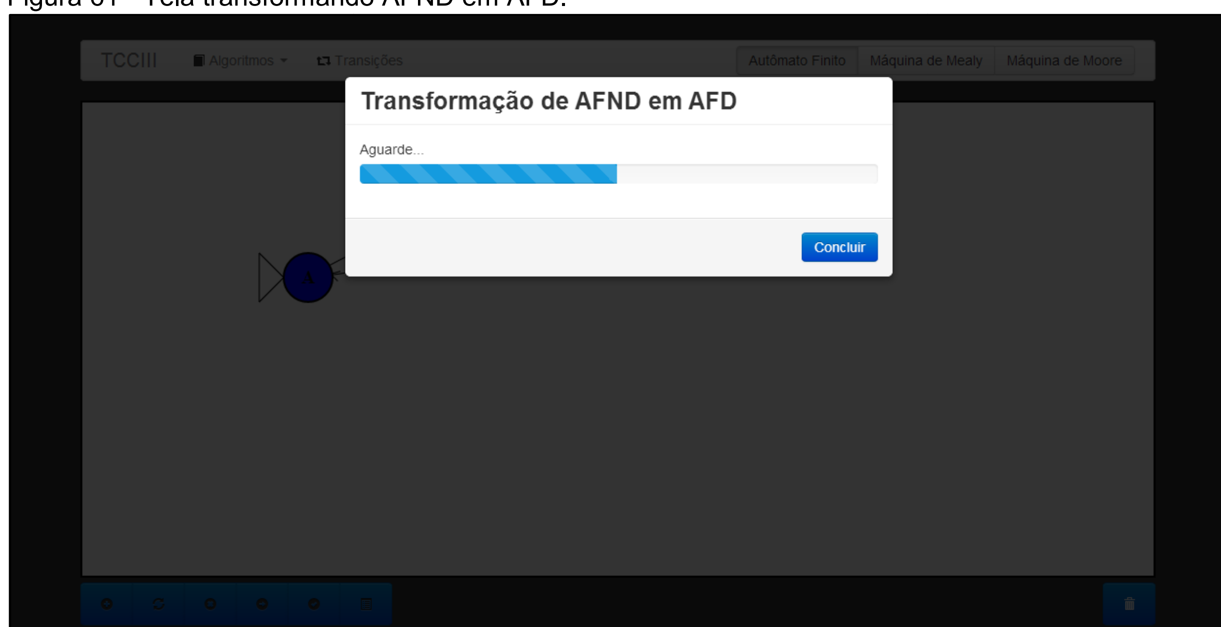


Fonte: Do autor.

Para testar uma sentença primeiramente deve ser criado um autômato finito. É obrigatório esse AF ter um estado inicial e pelo menos um estado final, se não é mostrada uma mensagem de erro informando a falta destes requisitos. Depois de criado o AF o usuário seleciona a opção reconhecedor de sentenças, onde terá um campo para informar a sentença e um botão para chamar a função que reconhece a sentença, como mostra a figura 60.

A transformação de um AFND em AFD não apresenta uma tela para o usuário, pois quando utilizada já retorna o AFD desenhado, mostrando apenas o progresso como segue na figura 61. O mesmo é aplicado quando é feita a minimização de um autômato finito.

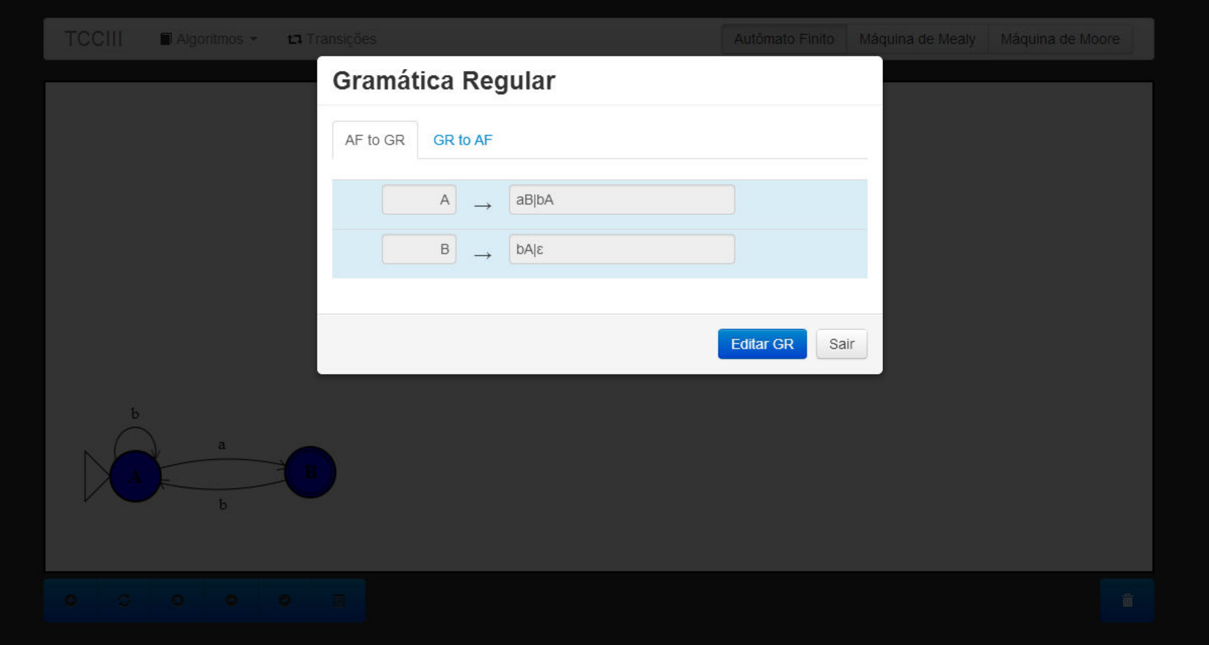
Figura 61 - Tela transformando AFND em AFD.



Fonte: Do autor.

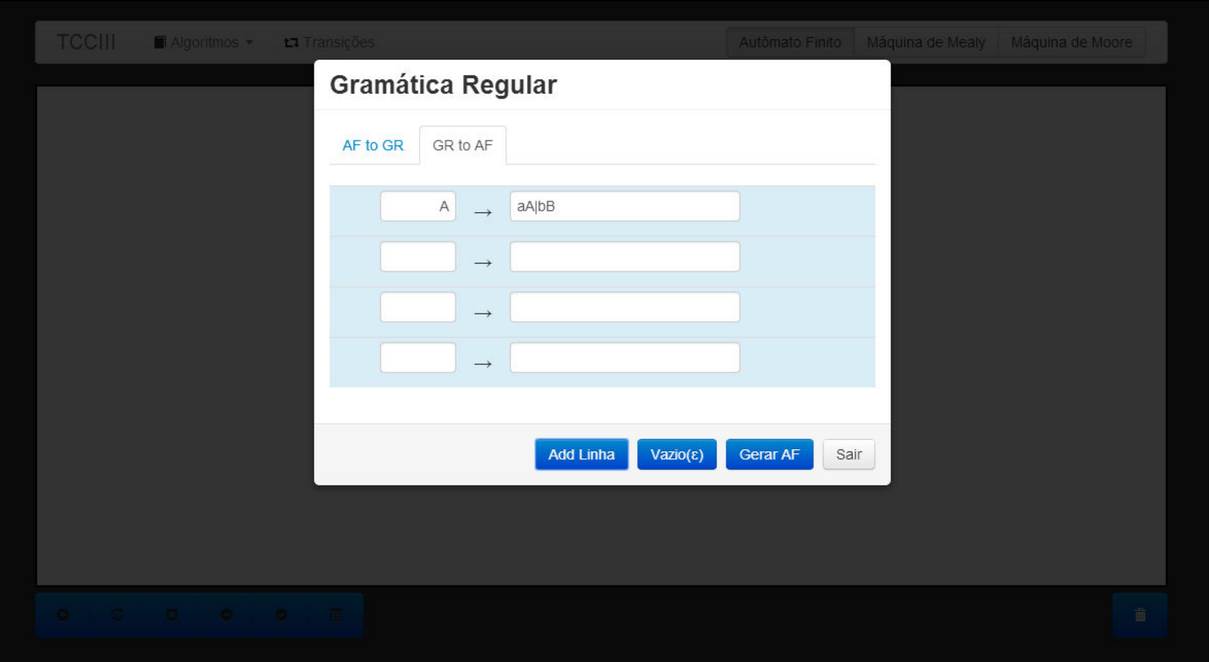
A figura 62 ilustra a tela de gramáticas regularas, onde o usuário terá acesso a duas abas. A primeira aba “AF to GR” é onde poderá ser visualizada a gramática regular gerada a partir do AF desenhado. Nesta aba também apresenta uma opção de editar a GR gerada, passando assim os valores para outra aba, e possibilitando o usuário incrementar o AF através de sua GR. Já a aba “GR to AF” (figura 63) serve para inserir uma GR. Apresenta as opções adicionar mais linha na gramática, inserir sinal de vazio no campo selecionado e gerar o AF equivalente a GR informada.

Figura 62 - Tela gramática regular - AF to GR.



Fonte: Do autor.

Figura 63 - Tela gramática regular - GR to AF.



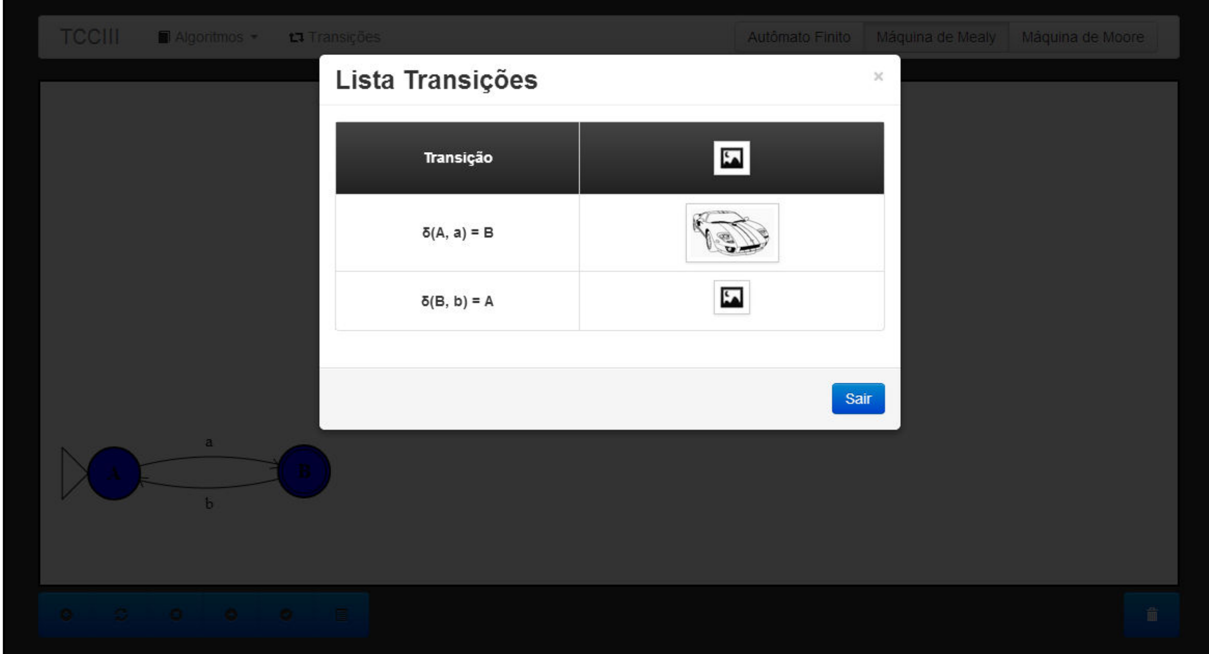
Fonte: Do autor.

Na inserção da gramática regular o usuário deve colocar todos não terminais em maiúsculo e os terminais em minúsculo. As produções devem ser separadas pelo caractere “|”, como pode ser visto na figura 63.

O módulo da máquina de Mealy e Moore possibilita o usuário simular as saídas de imagens associadas às transições (Mealy) ou estado (Moore). Na MMe é

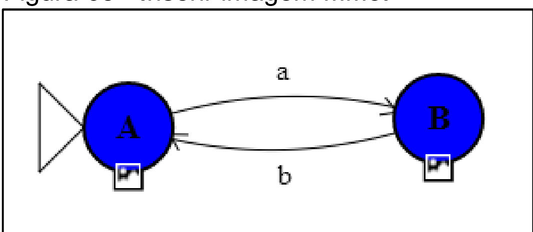
inserido uma opção no campo transições (figura 64) onde pode ser escolhida uma imagem no formato .jpg ou .png que será vinculada a transição. Para simular a máquina deve ser inserida uma sentença, onde será simulada na máquina apresentado a sequência de saída de imagem para o usuário, como mostra a figura 66. Se posicionar o mouse em cima das imagens é exibido qual estado esta relacionada. O mesmo serve para o módulo da máquina de Moore, mas nessa a imagem está vinculada ao estado, tendo uma opção no próprio desenho do estado para inserir a imagem (Figura 65).

Figura 64 - Inserir imagem MMe.



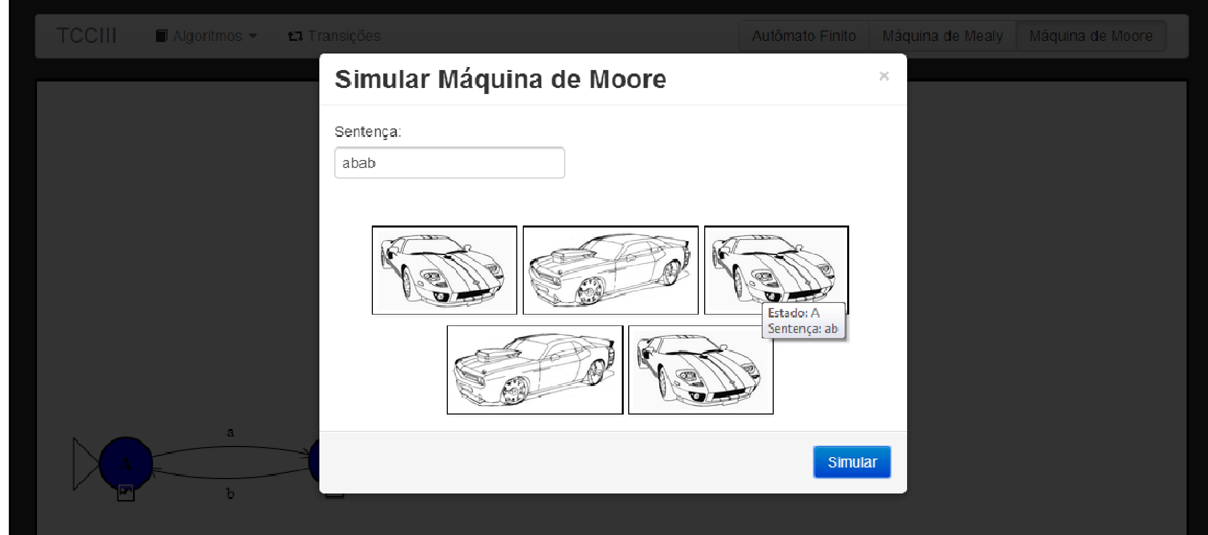
Fonte: Do autor.

Figura 65 - Inserir imagem MMo.



Fonte: Do autor.

Figura 66 - Tela simulação autômatos com saída.



Fonte: Do autor.

Por fim, observa-se que os módulos do AFLAB foram unidos em três módulos diferentes contendo funcionalidades que cada um manipula.

7 CONCLUSÃO

Hoje em dia diversas áreas de ensino utilizam ferramentas que aguçam e facilitam o conhecimento. Não é diferente na área de linguagens formais, onde através de trabalhos de conclusão de cursos da Universidade do Extremo Sul Catarinense foi disponibilizado um software que auxilia o ensino de autômatos finitos e gramáticas regulares, chamado AFLAB. Com o mesmo pensamento foi desenvolvido como objetivo deste trabalho de pesquisa um aplicativo WEB para criação e manipulação de autômatos finitos e gramáticas regulares, usando como base os algoritmos implementados no AFLAB.

Durante o desenvolvimento foi possível ter uma experiência na criação de uma aplicação WEB onde agregou conhecimentos na parte de HTML5, CSS e javascript. Foi notável também que a utilização dos frameworks bootstrap e kineticJs facilitaram bastante a implementação, tendo o bootstrap para o front-end da aplicação e o kineticJs na criação da tela de desenho.

Na fase de pesquisa deste trabalho foram encontradas algumas dificuldades, como a falta de conhecimento aprofundado em autômatos finitos e gramáticas regulares que resultaram em um exaustivo estudo, e também a pouca variedade de livros disponíveis sobre o assunto. Já na parte de desenvolvimento a maior dificuldade foi no ambiente de desenho, onde tiveram que ser realizado diversos cálculos para realizar a movimentação dos objetos na tela corretamente. Além disso, houve uma necessidade de um estudo sobre a utilização do elemento canvas do HTML5 e do framework KineticJS. Também foi encontrado dificuldade na implementação do método de minimização de AF, por ser bastante complexo de implementar e também por apresentar mais de uma forma de realizar onde acabavam confundindo.

Em relação aos algoritmos do AFLAB, serviram somente como um auxilio para implemetação de novos algoritmos, onde possibilitou fazer uma relação dos métodos na parte teórica e eles implementados em uma linguagem de programação. Foi optado pelo desenvolvimento de novos algoritmos devido à diferença na estrutura de armazenamento do AFLAB em relação à utilizada na aplicação WEB, onde precisava trabalhar com informações inseridas no canvas, necessitando assim criar outro método de percorrer e retornar as variáveis no algoritmo.

Foi possível então disponibilizar através da utilização dos frameworks Bootstrap e KineticJS, junto com o JAVASCRIPT uma ferramenta de ensino via WEB, com um ambiente bastante interativo e de fácil utilização, que não restringe o uso somente em sala de aula e pode ser utilizada por qualquer pessoa. Pode-se dizer que a aplicação desenvolvida é um aperfeiçoamento ou até mesmo uma atualização para os dias atuais da ferramenta AFLAB. Após todos os estudos aplicados e diversos testes realizados foi possível constatar que todos os objetivos desta pesquisa foram alcançados positivamente.

Por fim, é recomendado como trabalhos futuros a inclusão das expressões regulares, que é outro assunto de linguagens formais que tem total ligação com os autômatos finitos. Seria interessante também detectar se uma gramática é uma GR, desenhar um AF através de sua inserção na forma tabular e mostrar no ambiente de desenho a execução de cada algoritmo da aplicação, por exemplo, quando testar uma sentença mostrar o estado atual em outra cor e a cada transição mudar a cor da aresta exibindo qual transição esta fazendo naquele momento, mostrando para o usuário na forma visual os estados percorridos e as transições efetuadas durante a execução do algoritmo.

REFERÊNCIAS

BOVET, J. **Visual Automata Simulator: a tool for simulating automata and turing machines**. Universidade de San Francisco, 2004. Disponível em: <http://www.cs.usfca.edu/~jbovet/vas.html>. Acesso em: 25 nov. 2012.

FURTADO, Olinto José Varela. **Linguagens formais e compiladores**. Apostila da disciplina de Linguagens Formais e Compiladores da UFSC, 2004.

GAIDZINSKI, Marco Aurélio. **Ambiente de criação e manipulação de autômatos finitos na forma gráfica ou tabular para o reconhecimento de sentenças**. 2007. p. 61. Trabalho de Conclusão de Curso (Graduação). UNESC, Criciúma.

HOPCROFT, John E.; ULLMAN, Jeffrey D.; MOTWANI, Rajeev. **. Introdução à teoria de autômatos, linguagens e computação**. Tradução de Vandenberg D. de Souza. Rio de Janeiro: Elsevier, 2003.

LEWIS, Harry R.; PAPADIMITRIOU, Christos H. **. Elementos de teoria da computação**. Tradução de Edson Furmankiewicz. 2. ed. Porto Alegre: Bookman, 2004.

MENEZES, Paulo Fernando Blauth. **Linguagens formais e autômatos.** 5.ed Porto Alegre: Sagra Luzzatto, 2005.

OLIVEIRA, Marlon de Matos. **Autômatos finitos com saída: Um ambiente de criação e manipulação de maquinas de Moore e Mealy no AFLAB**. 2008. p. 82. Trabalho de Conclusão de Curso (Graduação). UNESC, Criciúma.

PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. **Implementação de linguagens de programação: compiladores**. Porto Alegre: Sagra Luzzatto, 2000.

RAMOS, Marcus Vinícius M.; NETO, João José; VEGA, Ítalo Santiago. **Linguagens Formais: Teoria, Modelagem e Implementação**. Porto Alegre: Bookman Companhia Editora, 2009.

RIBEIRO JUNIOR, Neilton Gonçalves.**Simulador de Autômatos**. Universidade de Uberaba, UNIUBE, 2009. Disponível em: < http://www.simuladordeautomatos.com>. Acesso em: 25 nov. 2012.

RODGER, Susan H. **Java Formal Language and Automata Package (JFLAP)**. Rensselaer Polytechnic Institute, RPI, Estados Unidos, 1990. Disponível em: <http://www.jflap.org>. Acesso em: 25 nov. 2012.

ROSA, João Luís Garcia. **Linguagens formais e autômatos**. Rio de Janeiro: LTC,

2010.

SERAFIN, Joelson Perdoná. **Manipulação de autômatos finitos no AFLAB**. 2009. p. 71. Trabalho de Conclusão de Curso (Graduação). UNESC, Criciúma.

SZYMANSKI, Charbel. **Ambiente para a construção e simulação de autômatos finitos**. Projeto de Trabalho de Conclusão de Curso em Ciência da Computação. Universidade do Sul de Santa Catarina, Tubarão, 2005.

TEIXEIRA, Aline. **Gramática Regulares e Expressões Regulares: Um ambiente de manipulação no AFLAB**. 2008. p. 83. Trabalho de Conclusão de Curso (Graduação). UNESC, Criciúma.

VIEIRA, Luiz Filipe Menezes; VIEIRA, Marcos Augusto Menezes; VIEIRA, Newton José. **Language Emulator, uma ferramenta de auxílio no ensino de Teoria da Computação**. In: II Workshop de Educação em Computação e Informática do Estado de Minas Gerais, 2003, Poços de Caldas, MG. Disponível em: <http://homepages.dcc.ufmg.br/~mmvieira/publications/weimig-languageEmulator.pdf>. Acesso em: 25 nov. 2012.

VIEIRA, Newton José. **Introdução aos Fundamentos da Computação: Linguagens e Máquinas**. São Paulo: Pioneira Thomson Learning, 2006.

APÊNDICE A - ARTIGO

AF-GR WEB: UM APLICATIVO WEB, PARA MANIPULAÇÃO DE AUTÔMATOS FINITOS E GRAMÁTICAS REGULARES

Diego Possebon Fernandes[1], Christine Vieira[1]

[1]Ciência da Computação – Universidade do Extremo do Sul Catarinense(UNESC)
Caixa Postal 3167 – 88.806-000 – Criciúma – SC – Brasil

diegopossebonf@gmail.com, cvi@unesc.net

Abstract. *This article presents the development of a web application to handlee of finite automata and regular grammars, based on the algorithm used in the AFLAB software. Due to the presented problems in the AFLAB software use was proposed the development of a new app, which intends to solve these problems, to integrate the modules that did not integrate in AFLAB and to available an interactive grafic to manipulate the elements. Lastly, we have a web application with a highly interactive ambience and easily use, that is like a support in the formal languages study.*

Resumo. *Este artigo apresenta o desenvolvimento de uma aplicação WEB para manipulação de autômatos finitos e gramáticas regulares, tendo como base os algoritmos utilizados no software AFLAB. Devido a problemas apresentados na utilização no software AFLAB foi proposto o desenvolvimento de um novo aplicativo, que busca solucionar esses problemas, integrar os módulos que não se integravam no AFLAB e disponibilizar uma interface gráfica interativa para manipulação dos elementos. Por fim, temos uma aplicação WEB com um ambiente bastante interativo e de fácil utilização, que serve como um apoio no estudo de linguagens formais.*

1. Introdução

A teoria das Linguagens Formais (LF) é parte da teoria da computação e, com tal, é imprescindível seu conhecimento por todos os profissionais e acadêmicos da área (ROSA, 2010). Isso acontece por que essa teoria é aplicada em análise léxica e análise sintática de diversas linguagens de programação utilizadas atualmente.

Um assunto abordado em LF é Autômato Finito (AF). O AF é um sistema de estados finito o qual constitui um modelo computacional do tipo sequencial muito comum em estudos de linguagens formais, compiladores, semântica formal e modelos para concorrência. Em LF este sistema é utilizado como reconhecedor de linguagens, que recebe como entrada uma cadeia de símbolos com o objetivo de mostrar se essa cadeia faz parte de determinada linguagem ou não. São classificados em Autômato Finito Determinístico (AFD) e Autômato Finito Não Determinístico (AFND). O AFD consiste que a partir de um estado corrente e do símbolo lido, o sistema pode assumir um único estado. Já o AFND pode assumir um conjunto de estados alternativos a partir de um estado e do símbolo de entrada. (MENEZES, 2005).

Uma extensão do AF são os autômatos finitos com saída denominados de máquina de Mealy e máquina de Moore. Com estas máquinas é possível gerar uma palavra de saída, ou seja, o autômato não fica limitado à aceita ou rejeita. Esses autômatos podem ser utilizados em aplicações como hipertexto, hipermídia e animação quadro-a-quadro, pois as saídas

podem ser diversos tipos de dados como imagens, músicas, textos, entre outros (HOPCROFT; ULLMAN; MOTWANI, 2003).

Linguagem formal abrange também assuntos como gramáticas regulares, expressões regulares, minimização de autômatos finitos, transformação de autômatos finitos não determinísticos em determinísticos, geração de gramática regular a partir do autômato finito determinístico, entre outros. Diversos assuntos que necessitam de uma série de exercícios para melhor compreensão do aluno. Estes exercícios na maioria da vezes acabam sendo feitos a lápis e caneta, limitando os alunos a exemplos não muito complexos. Com objetivo de auxiliar o aluno nesse aprendizado, foi criado o grupo de pesquisa de linguagens formais do curso de ciência da computação da UNESC, que desenvolveu o AFLAB, software de simulação de autômatos finitos.

Segundo a professora da disciplina de LF alguns problemas foram detectados com a utilização do software em sala de aula. O AFLAB foi sendo incrementado com novos módulos de acordo com outros trabalhos desenvolvidos, onde alguns módulos acabaram não sendo integrados entre si, e para que se tenha acesso a todos os módulos desenvolvidos é necessária à instalação de três versões diferentes. Na elaboração do autômato na forma gráfica, este não tem a opção de representar uma transição de um estado para ele mesmo, ou seja, um laço. Além disso, não tem as setas nas arestas. Foi constatada também uma dificuldade na inserção de uma gramática regular ou de um autômato na forma tabular, pois a tela para entrada dos dados é muito complexa para o usuário. Estes problemas acabaram inviabilizando a utilização da ferramenta em sala de aula.

A partir do que foi observado, é proposto à criação de um aplicativo WEB, para manipulação de Autômatos Finitos e de Gramáticas Regulares, tendo como base os algoritmos utilizados nos módulos do AFLAB. O aplicativo irá integrar os módulos buscando as resoluções dos problemas apresentados, e terá como foco uma interface mais interativa e de fácil utilização, para que se torne uma opção de utilização mais viável dentro e fora da sala de aula.

2. Autômatos Finitos

Um autômato finito consiste em um sistema de estados finitos, ou seja, um conjunto de estados finito e pré-definido onde cada estado tem apenas informações do passado, necessárias para definir as ações da próxima entrada. Podem ser definidos como reconhecedores de linguagens regulares ou expressões regulares, pois recebem uma sequência de caracteres como entrada, com o objetivo de conferir se a mesma pertence ou não à linguagem estabelecida (MENEZES, 2005).

Menezes (2005) define um autômato finito como uma máquina composta, basicamente, de três partes:

a) fita: dispositivo de entrada que contem a informação a ser processada;

b) unidade de controle: reflete o estado corrente da máquina. Possui uma unidade de leitura (cabeça da fita) a qual acessa uma célula da fita de cada vez e movimenta-se exclusivamente para a direita;

c) programa ou função de transição: função que comanda as leituras e define o estado corrente da máquina.

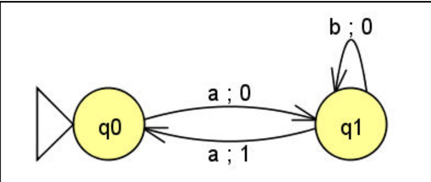
Os autômatos finitos possuem duas classes com distinções fundamentais, em que em uma o controle pode ser "determinístico", ou seja, não pode transitar para vários estados a

partir do estado atual, e outra que é não determinístico, onde pode transitar para mais de um estado a partir do estado atual (HOPCROFT; ULLMAN; MOTWANI, 2003).

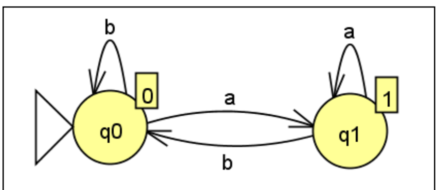
Em autômatos finitos ainda pode ser feito a transformação de um autômato finito não determinístico em um determinístico e a minimização que consiste em um autômato finito com o menor número de estados possíveis através da eliminação dos estados inúteis, inacessíveis e equivalentes.

2.1. Autômatos Finitos com Saída

Consiste em uma modificação no Autômato Finito que possibilita gerar saídas. Essas saídas podem ser relacionadas às transições (máquina de Mealy) ou aos estados (máquina de Moore).



2 Figura 1. Máquina de Mealy



3 Figura 2. Máquina de Moore

3. Gramática Regular

As gramáticas regulares (GR) fornecem vários modos de determinar uma linguagem regular. Os autômatos finitos apresentados nos capítulos anteriores permitem a especificação de uma linguagem através de um reconhecedor para a mesma, já as gramáticas regulares permitem especificação através de um gerador de linguagem, ou seja, mediante a uma gramática regular, aponta como gerar todas, e apenas, as palavras de uma linguagem regular (VIEIRA, 2006). Segue abaixo um exemplo de uma GR:

$$S \rightarrow aA \mid bA \mid \varepsilon$$

$$A \rightarrow aS \mid bS$$

As gramáticas regulares geram apenas linguagens regulares e, vice-versa. Com isso através de uma GR podemos gerar um AF que reconhece a linguagem da GR, e através de um AF podemos gerar a GR referente à linguagem que o AF reconhece.

4. AF-GR WEB

Este capítulo apresenta o desenvolvimento da aplicação WEB proposta como objetivo deste trabalho. A aplicação disponibiliza um ambiente gráfico de fácil utilização, onde podem ser criados autômatos finitos na forma gráfica através de botões com finalidades especificas e ações com o mouse na tela. Há três módulos de criação diferentes na aplicação com suas respectivas funcionalidades, onde inicialmente temos o módulo de autômatos finitos, que pode ser testado uma sentença, transformar um AFND em AFD, fazer a minimização, gerar a gramática regular através do AF desenhado e desenhar um AF através da inclusão de uma

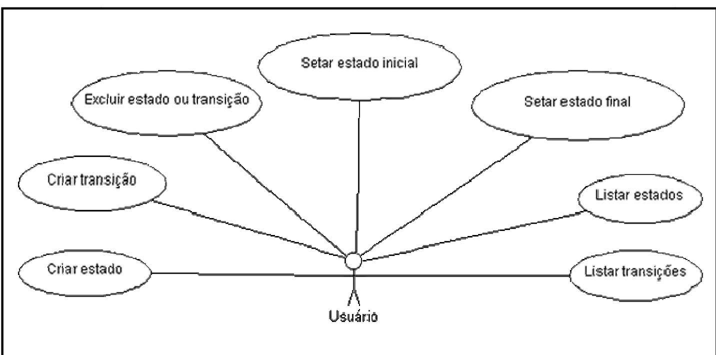
gramática. Nos outros dois módulos podem ser simulados máquina de Mealy ou Moore, onde temos imagens como tipo de saída associadas às transições (Mealy) ou aos estados (Moore).

4.1. Metodologia

Para atingir os objetivos propostos neste projeto de pesquisa foram necessárias algumas etapas metodológicas, onde primeiramente foi feito um levantamento bibliográfico, buscando um melhor conhecimento das linguagens formais, com o foco voltado para os autômatos finitos e as gramáticas regulares. Foi feito um estudo sobre os AFD e AFND, suas diferenças e a forma de transformação de AFND em AFD. O modo de minimizar um AF, autômatos finitos com saída (máquina de Mealy e Moore), como gerar um AF através de uma GR e uma GR através de um AF também estiveram neste estudo. Por fim, foi disponibilizado o código fonte do aplicativo AFLAB, que complementou esse estudo através da compreensão dos seus algoritmos já implementados.

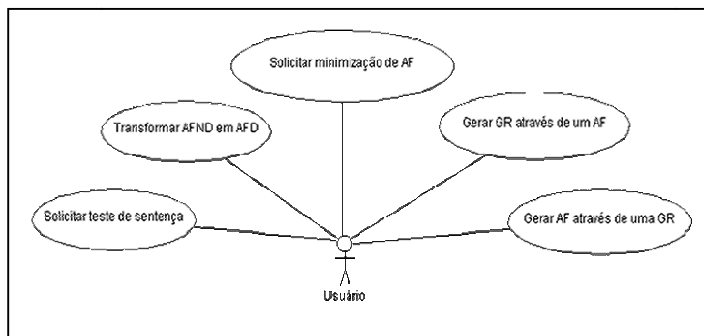
Foi de grande importância também um pequeno estudo da utilização dos frameworks Bootstrap e KineticJS para utilizar no desenvolvimento da aplicação. O HTML5, CSS e javascript foram escolhidos por serem ferramentas mais utilizadas nos aplicativos WEB atualmente.

Após o levantamento bibliográfico foi dado início ao desenvolvimento do aplicativo WEB. Primeiramente foi feito a modelagem UML, onde foram obtidos alguns diagramas.

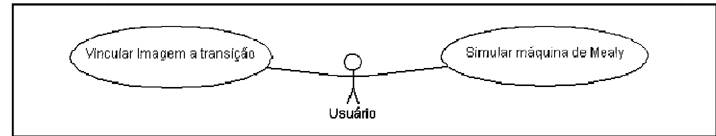


4 Figura 3. Diagrama de caso de uso (Usuário).

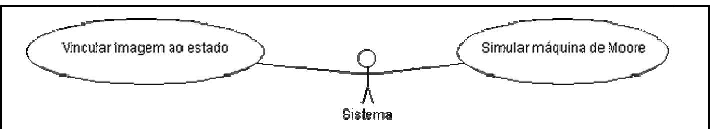
A figura 3 representa o diagrama de caso de uso genérico para todos os módulos, ou seja, são funcionalidades acessíveis em todos os módulos do aplicativo. As figuras 4, 5, 6 representam os diagramas das funcionalidades por módulo.



5 Figura 4. Diagrama de caso de uso (Usuário) - módulo autômato finito.



6 **Figura 5. Diagrama de caso de uso (Usuário) - módulo máquina de Mealy.**



7 **Figura 6. Diagrama de caso de uso (Usuário) - módulo máquina de Moore.**

4.2. Desenvolvimento

No desenvolvimento da aplicação foi utilizado o HTML5 na criação das páginas web, junto com a linguagem CSS para estilizar as páginas. Para facilitar o desenvolvimento foi utilizado o Bootstrap, que é um framework front-end que disponibiliza uma coleção de vários elementos e funções personalizáveis para projetos web. O ambiente gráfico foi desenvolvido no elemento canvas do HTML5 utilizando o framework KineticJS, que permite desenhar formas e imagens no canvas com mais facilidade. E por fim, foi utilizado o javascript para fazer as ações do usuário com os elementos da página.

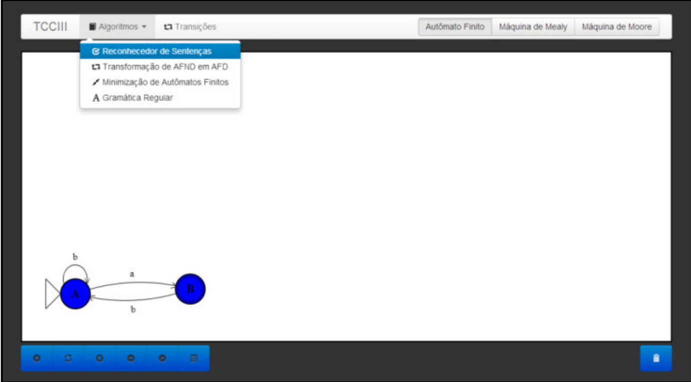
Como estrutura de armazenamento do aplicativo foi definida duas classes: Estado e Transição. As classes são armazenadas em dois arrays dinâmicos, sendo um a lista de estados e outro a lista de transições.

O aplicativo foi dividido em três módulos, cada um com suas respectivas funcionalidade. Os módulos são autômatos finitos, máquina de Mealy e máquina de Moore.

5. Resultados Obtidos

A aplicação disponibiliza um ambiente gráfico para a criação dos autômatos finitos, onde o usuário poderá introduzir o AF que ira tratar. Inicialmente é necessário selecionar no canto superior direito da aplicação um dos três tipos de AF a ser desenhado. O tipo autômato finito já vem por padrão e se for desenhar uma maquina de Mealy ou Moore, o tipo específico deve ser selecionado.

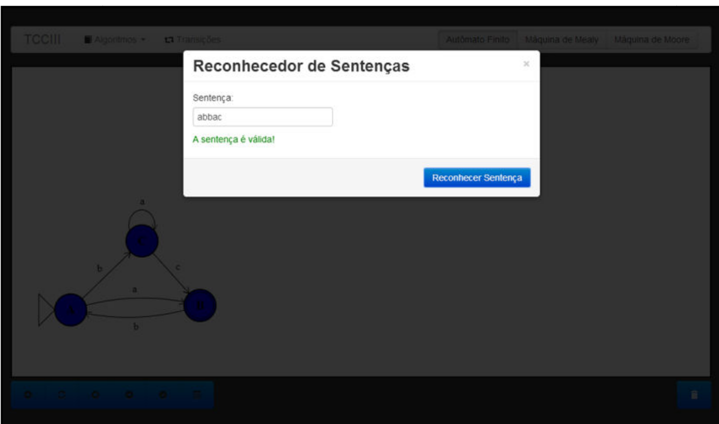
Após selecionar o tipo, o AF já pode ser desenhado através das opções de criação logo abaixo da tela de desenho, no canto inferior esquerdo. As opções são criar estado, criar transição, excluir, definir estado como estado inicial, definir estados finais e listar estados. Para criar um estado deve ser selecionada a opção referente à criação de estado e após clicar na tela de desenho, onde cada clique desenha um estado. Já para criar uma transição segue a mesma lógica anteriormente, mas agora deve clicar e segurar o botão esquerdo do mouse em cima do estado que vai ser origem da transição e após arrastar até o estado destino e largar o botão, mostrando assim à opção de inserir um símbolo para a transição. Nas opções de definir um estado inicial ou estado finais basta clicar em cima do estado desejado. Por fim tem a opção de listar os estados do AF onde mostra o nome, se é estado inicial e final. A figura 7 apresenta o ambiente gráfico.



8 Figura 7. Ambiente gráfico

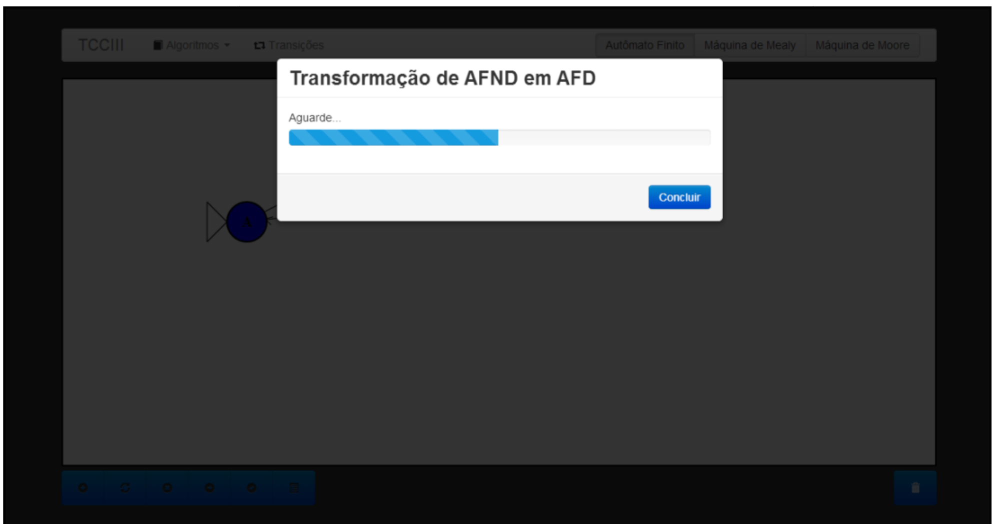
No cabeçalho da página tem a opção algoritmo (figura 7), que como o nome já diz, disponibiliza os algoritmos referentes ao tipo selecionado, ou seja, se tiver selecionado a máquina de Moore, aparecerá somente os algoritmos aplicados a esta opção. Seguindo no cabeçalho tem a opção transições, onde lista todas as transições criadas. Se tiver selecionado para desenhar maquina de Mealy, a lista de transições terá mais uma coluna onde será utilizada para associar saída para as transições. Por fim tem o botão que limpa toda tela, localizado no canto inferior direito.

Para testar uma sentença primeiramente deve ser criado um autômato finito. É obrigatório esse AF ter um estado inicial e pelo menos um estado final, se não é mostrada uma mensagem de erro informando a falta destes requisitos. Depois de criado o AF o usuário seleciona a opção reconhecedor de sentenças, onde terá um campo para informar a sentença e um botão para chamar a função que reconhece a sentença, como mostra a figura 8.



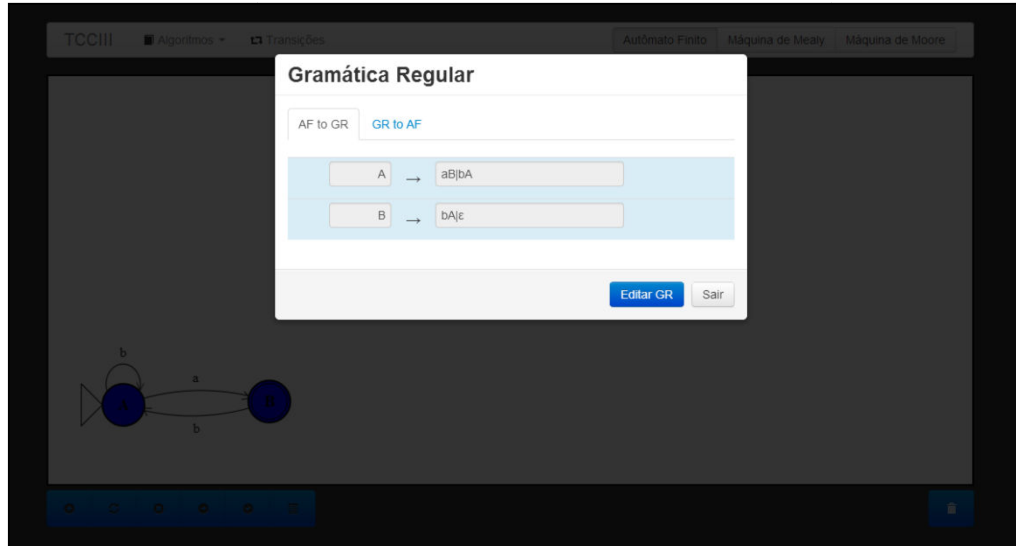
9 Figura 8. Reconhecedor de sentenças

A transformação de um AFND em AFD não apresenta uma tela para o usuário, pois quando utilizada já retorna o AFD desenhado, mostrando apenas o progresso como segue na figura 9. O mesmo é aplicado quando é feita a minimização de um autômato finito.

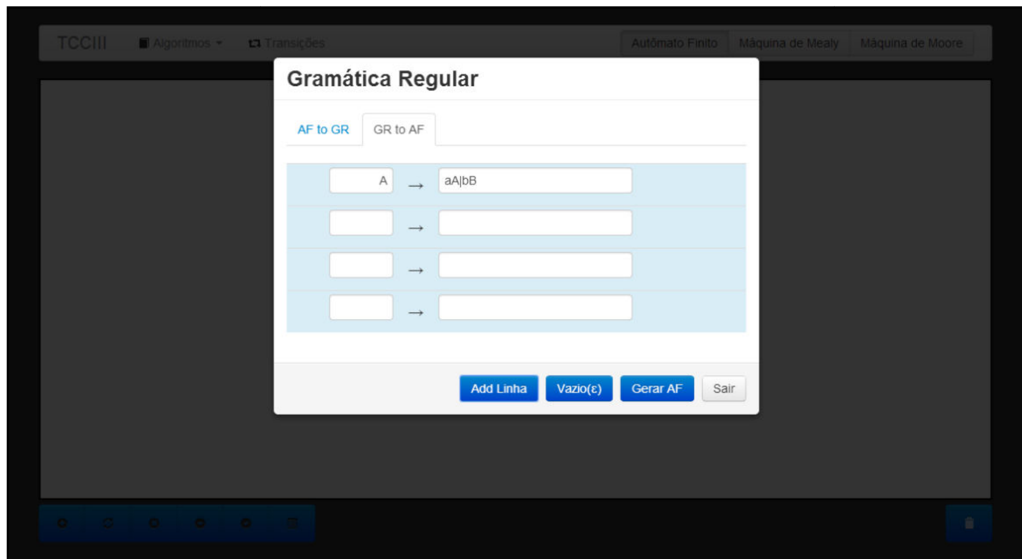


10 Figura 9. Tela transformação AFND em AFD.

A figura 10 ilustra a tela de gramáticas regularas, onde o usuário terá acesso a duas abas. A primeira aba "AF to GR" é onde poderá ser visualizada a gramática regular gerada a partir do AF desenhado. Nesta aba também apresenta uma opção de editar a GR gerada, passando assim os valores para outra aba, e possibilitando o usuário incrementar o AF através de sua GR. Já a aba "GR to AF" (figura 11) serve para inserir uma GR. Apresenta as opções adicionar mais linha na gramática, inserir sinal de vazio no campo selecionado e gerar o AF equivalente a GR informada.



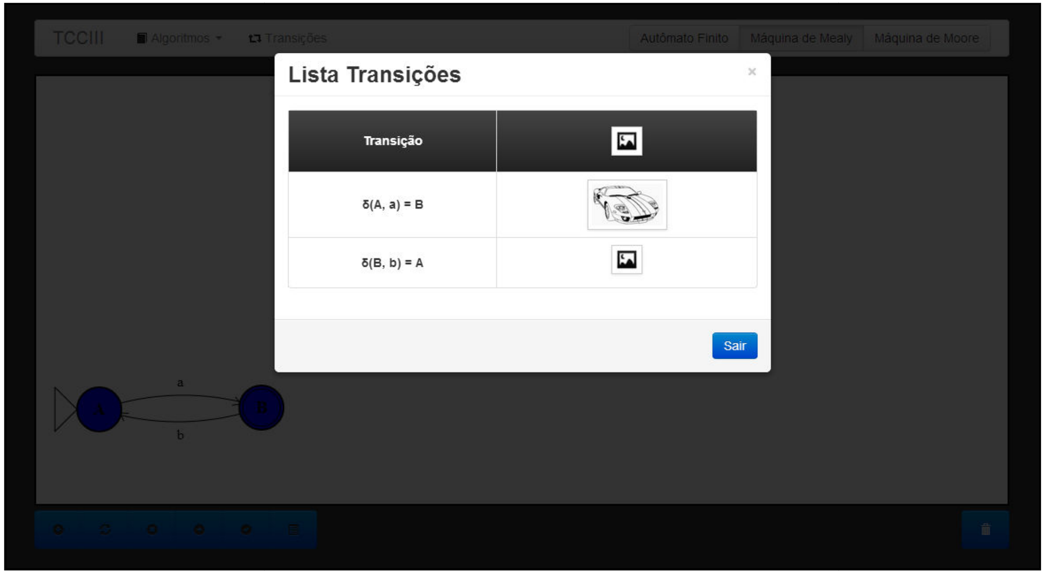
11 Figura 10. Tela gramática regular - AF to GR



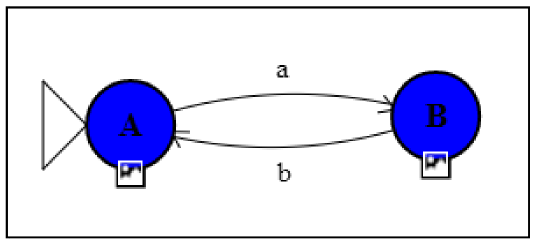
12 Figura 11. Tela gramática regular - GR to AF

Na inserção da gramática regular o usuário deve colocar todos não terminais em maiúsculo e os terminais em minúsculo. As produções devem ser separadas pelo caractere “|”, como pode ser visto na figura 11.

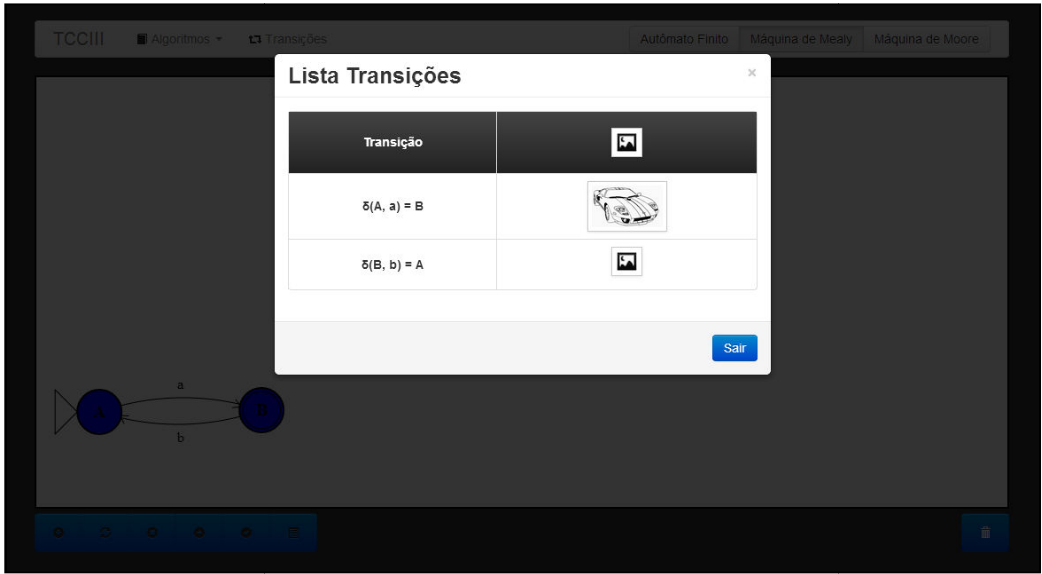
O módulo da máquina de Mealy e Moore possibilita o usuário simular as saídas de imagens associadas às transições (Mealy) ou estado (Moore). Na MMe é inserido uma opção no campo transições (figura 12) onde pode ser escolhida uma imagem no formato .jpg ou .png que será vinculada a transição. Para simular a máquina deve ser inserida uma sentença, onde será simulada na máquina apresentado a sequência de saída de imagem para o usuário, como mostra a figura 14. Se posicionar o mouse em cima das imagens é exibido qual estado esta relacionada. O mesmo serve para o módulo da máquina de Moore, mas nessa a imagem está vinculada ao estado, tendo uma opção no próprio desenho do estado para inserir a imagem (Figura 13).



13 Figura 12. Inserir imagem MMe



14 Figura 13. Inserir imagem MMo



15 Figura 14. Tela simulação autômatos com saída

Por fim, observa-se que os módulos do AFLAB foram unidos em três módulos diferentes contendo funcionalidades que cada um manipula.

6. Conclusão

Hoje em dia diversas áreas de ensino utilizam ferramentas que aguçam e facilitam o conhecimento. Não é diferente na área de linguagens formais, onde através de trabalhos de conclusão de cursos da Universidade do Extremo Sul Catarinense foi disponibilizado um software que auxilia o ensino de autômatos finitos e gramáticas regulares, chamado AFLAB. Com o mesmo pensamento foi desenvolvido como objetivo deste trabalho de pesquisa um aplicativo WEB para criação e manipulação de autômatos finitos e gramáticas regulares, usando como base os algoritmos implementados no AFLAB.

Durante o desenvolvimento foi possível ter uma experiência na criação de uma aplicação WEB onde agregou conhecimentos na parte de HTML5, CSS e javascript. Foi notável também que a utilização dos frameworks bootstrap e kineticJs facilitaram bastante a implementação, tendo o bootstrap para o front-end da aplicação e o kineticJs na criação da tela de desenho.

Na fase de pesquisa deste trabalho foram encontradas algumas dificuldades, como a falta de conhecimento aprofundado em autômatos finitos e gramáticas regulares que resultaram em um exaustivo estudo, e também a pouca variedade de livros disponíveis sobre o assunto. Já na parte de desenvolvimento a maior dificuldade foi no ambiente de desenho, onde tiveram que ser realizado diversos cálculos para realizar a movimentação dos objetos na tela corretamente. Além disso, houve uma necessidade de um estudo sobre a utilização do elemento canvas do HTML5 e do framework KineticJS. Também foi encontrado dificuldade na implementação do método de minimização de AF, por ser bastante complexo de

implementar e também por apresentar mais de uma forma de realizar onde acabavam confundindo.

Em relação aos algoritmos do AFLAB, serviram somente como um auxilio para implemetação de novos algoritmos, onde possibilitou fazer uma relação dos métodos na parte teórica e eles implementados em uma linguagem de programação. Foi optado pelo desenvolvimento de novos algoritmos devido a diferença na estrutura de armazenamento do AFLAB em relação à utilizada na aplicação WEB, onde precisava trabalhar com informações inseridas no canvas, necessitando assim criar outro método de percorrer e retornar as variáveis no algoritmo.

Foi possível então disponibilizar através da utilização dos frameworks Bootstrap e KineticJS, junto com o JAVASCRIPT uma ferramenta de ensino via WEB, com um ambiente bastante interativo e de fácil utilização, que não restringe o uso somente em sala de aula e pode ser utilizada por qualquer pessoa. Pode-se dizer que a aplicação desenvolvida é um aperfeiçoamento ou até mesmo uma atualização para os dias atuais da ferramenta AFLAB. Após todos os estudos aplicados e diversos testes realizados foi possível constatar que todos os objetivos desta pesquisa foram alcançados positivamente.

Por fim, é recomendado como trabalhos futuros a inclusão das expressões regulares, que é outro assunto de linguagens formais que tem total ligação com os autômatos finitos. Seria interessante também detectar se uma gramática é uma GR, desenhar um AF através de sua inserção na forma tabular e mostrar no ambiente de desenho a execução de cada algoritmo da aplicação, por exemplo, quando testar uma sentença mostrar o estado atual em outra cor e a cada transição mudar a cor da aresta exibindo qual transição esta fazendo naquele momento, mostrando para o usuário na forma visual os estados percorridos e as transições efetuadas durante a execução do algoritmo.

7. References

Menezes, Paulo Fernando Blauth. (2005) Linguagens formais e autômatos. 5.ed. Porto Alegre: Sagra Luzzatto.

Rosa, João Luís Garcia. (2010) Linguagens formais e autômatos. Rio de Janeiro: LTC.

Vieira, Newton José. (2006) Introdução aos Fundamentos da Computação: Linguagens e Máquinas. São Paulo: Pioneira Thomson Learning.

Hopcroft, John E., Ullman, Jeffrey D., Motwani, Rajeev. (2003) Introdução à teoria de autômatos, linguagens e computação. Tradução de Vandenberg D. de Souza. Rio de Janeiro: Elsevier.