SiGrAλe una herramienta para la enseñanza de Teoría de Autómatas y Lenguajes

Ingrid Rocío Baquero Villamil
Técnologa en Sistematización de Datos

Diego David Satoba Castro
Técnologo en Administración de Sistemas

Politécnico Grancolombiano Institución Universitaria
Facultad de Ingeniería y Ciencias Básicas
Departamento de Sistemas
Bogotá, D.C.
Julio de 2011

SiGrAλe una herramienta para la enseñanza de Teoría de Autómatas y Lenguajes

Ingrid Rocío Baquero Villamil
Técnologa en Sistematización de Datos

Diego David Satoba Castro
Técnologo en Administración de Sistemas

Trabajo de tesis para optar al título de
Ingeniero de Sistemas

Director
Rafael Armando García Gomez, M.Sc.
Maestro en Matemáticas

Politécnico Grancolombiano Institución Universitaria
Facultad de Ingeniería y Ciencias Básicas
Departamento de Sistemas
Bogotá, D.C.
Julio de 2011

Nota de aceptación

Trabajo de tesis

"Mención "

Jurado
Diego Arevalo Ovalle

Jurado
Edwin Andrés Niño Velásquez

Director
Rafael Armando García Gomez

Bogota, D.C., Julio 28 de 2011

Índice general

Resumen

La teoría de autómatas y lenguajes brinda un panorama de diversos modelos computacionales que son la base a la hora de realizar compiladores para los lenguajes de programación.

Un curso convencional de teoría de autómatas y lenguajes está dirigido a estudiantes de ingeniería de sistemas, matemáticas y/o carreras afines, que previamente han adquirido una formación académica en programación, matemáticas y algoritmos. Se considera este curso como una introducción a otros más avanzados como diseño y fundamentos de compiladores, computabilidad y teoría de la complejidad algorítmica.

En el Politécnico Grancolombiano y en otras instituciones de educación superior generalmente el curso se compone tanto de una parte teórica como de una práctica, por lo cual es de gran utilidad contar con herramientas computacionales que apoyen la enseñanza del curso, y permitan a los estudiantes tener una mejor compresión de los conceptos vistos en clase.

Aunque ya existen algunas herramientas computacionales, estas no pueden ser incluidas dentro del material de un curso debido a las restricciones que se presentan en las licencias de los diferentes software. Por estas limitaciones surge la necesidad de diseñar una herramienta computacional con un ambiente gráfico, que pueda ser incluida como material académico y facilite la enseñanza, en partícular, del curso de autómatas, gramáticas y lenguajes que hace parte del pensum de la carrera de ingeniería de sistemas en el Politécnico Grancolombiano.

Este documento se encuentra dividido en varios capítulos. En el capítulo uno se encuentran los conceptos básicos relacionados con teoría de autómatas y lenguajes, los algoritmos implementados y sus respectivas demostraciónes. El capítulo dos presenta la herramienta `SiGrAλe`, sus características y funcionalidades, descripción de las ventajas del código y su estructura y en el capítulo tres se exponen los resultados obtenidos de la investigación y el desarrollo de la herramienta `SiGrAλe`.

Objetivos

Objetivo General

Desarrollar una herramienta grafica de teoría de autómatas y lenguajes, que pueda ser incluida como material académico para que apoye la enseñanza del curso de autómatas, gramáticas y lenguajes del Politécnico Grancolombiano.

Objetivos Específicos

- Implementar una herramienta que comprenda el diseño de automátas finitos.
- La herramienta debe ser capaz de realizar operaciones entre autómtas finitos.
- Implementar algoritmos de conversión de AFND a AFD.
- Implementar el algoritmo de minimización de estados de un autómata finito
- Implementar el algoritmo de equivalencia entre autómatas finitos.
- La herramienta debe ser capaz de generar autómata finito a partir de una expresión regular.
- Realizar la simulación para la evaluación de una cadena.

CAPÍTULO 1

Conceptos Preliminares

Una herramienta computacional para la enseñanza de la teoría de autómatas y lenguajes presupone la implementación de una serie de conceptos ligados a través de resultados teóricos que constituyen la teoría de autómatas y son establecidas como las pautas para el desarrollo de la herramienta `SiGrAλe`. En este capítulo presentamos los conceptos fundamentales que subyacen a la *teoría de autómatas* comenzando con una breve discusion acerca de la naturaleza de los lenguajes manipulables, definiendo primero lo que son *alfabetos* en computación. A partir de la definición de un lenguaje se muestran las operaciones entre lenguajes distinguiendo la representación de *lenguajes regulares* por medio de *expresiones regulares*, por último se presentan los autómatas como herramientas de análisis de lenguajes regulares.

En capítulos posteriores se muestra cómo los conceptos expuestos aquí se traducen y constituyen la herramienta `SiGrAλe` desarrollada en el lenguaje de programación `Java`.

1.1. Lenguajes Formales

Un problema grande en computación es la identificación, análisis e interpretación del lenguaje natural. Computacionalmente hablando un lenguaje se puede reducir a una organización de símbolos en *cadenas* bajo reglas concretas de asociación[6], partiendo de esto y desde el punto de vista teórico el primer paso será la identificación de los elementos básicos sobre los cuales se construye un lenguaje.

1.1.1. Alfabetos y Cadenas

Un **alfabeto** Σ es un conjunto finito no vacío cuyos elementos se les denominan **símbolos**. Una **cadena** sobre un alfabeto es una secuencia finita de símbolos pertenecientes a este alfabeto. Si α es una cadena sobre el alfabeto Σ se representa su longitud con $|\alpha|$. La cadena de longitud de 0 es llamada la cadena vacía y es representada por λ. El conjunto de todas las cadenas sobre un alfabeto Σ , incluyendo la cadena vacía, se denota por Σ^*.

De manera alterna una cadena puede ser definida inductivamente de la siguiente manera:

1. $\lambda \in \Sigma^*$ es cadena.
2. $\alpha x \in \Sigma^*$, $\forall x \in \Sigma$ y $\forall \alpha \in \Sigma^*$

1.1.2. Concatenación de cadenas

La concatenación de cadenas se puede definir inductiva o recursivamente. Si α, $\beta \in \Sigma^*$, $\theta \in \Sigma$, entonces:

1. $\alpha \cdot \lambda = \lambda \cdot \alpha = \alpha$.
2. $\alpha \cdot (\beta\theta) = (\alpha \cdot \beta)\theta$.

Propiedad 1.1.2.1 (Concatenación)**.** La concatenación de cadenas es una operación asociativa.[3] Es decir, si α, β, $\theta \in \Sigma^*$, entonces

$$(\alpha\beta)\theta = \alpha(\beta\theta). \tag{1.1.1}$$

1.1.3. Lenguajes

Un **lenguaje** L sobre un alfabeto Σ es un subconjunto de Σ^*. Todo lenguaje L satisface $\emptyset \subseteq L \subseteq \Sigma^*$ y puede ser finito o infinito y se denotan con letras mayúsculas.[3]. Debido a que los lenguajes son en si conjuntos, específicamente subconjuntos de Σ^*, las operaciones usuales entre conjuntos también son validas entre lenguajes.

Los siguientes lenguajes se destacan entre todos los lenguajes:

1. $\emptyset$ es el lenguaje vacío.
2. $\{\lambda\}$ es el lenguaje vacío que consta unicamente de la cadena vacía (λ).
3. Σ^* es el lenguaje de todas las cadenas sobre el alfabeto Σ.

1.1.4. Concatenación de Lenguajes

La concatenación de dos lenguajes A y B, denotada comunmente como $A \circ B$ o AB se define como

$$A \circ B = (\alpha\beta : \alpha \in A, \beta \in B). \tag{1.1.2}$$

Propiedades de la concatenación de lenguajes. Sean A, B, C lenguajes sobre Σ, entonces se tiene las siguientes propiedades:

1. $A \circ \emptyset = \emptyset \circ A = \emptyset$
2. $A \circ \{\lambda\} = \{\lambda\} \circ A = A$

3. Propiedad Asociativa
$$A \circ (B \circ C) = (A \circ B) \circ C \tag{1.1.3}$$

4. Distributividad de la concatenación con respecto a la unión,
$$A \circ (B \cup C) = A \circ B \cup A \circ C \tag{1.1.4}$$
$$(B \cup C) \circ A = B \circ A \cup C \circ A \tag{1.1.5}$$

5. Propiedad distributiva generalizada. Si $\{B_i\}_{i \in I}$ es una familia cualquiera de lenguajes sobre Σ entonces
$$A \circ \bigcup_{i \in I} B_i = \bigcup_{i \in I} (A \circ B_i), \tag{1.1.6}$$
$$\left(\bigcup_{i \in I} B_i \right) \circ A = \bigcup_{i \in I} (B_i \circ A) \tag{1.1.7}$$

1.1.5. Operaciones entre lenguajes

Las operaciones entre lenguajes se conocen como `operaciones conjuntistas o booleanas`, para diferenciarlas de las operaciones lingüísticas. Dos lenguajes **A** y **B**, generan nuevos lenguajes a partir de las operaciones que se pueden realizar entre ellos.[3]

Sea **A** y **B** lenguajes sobre Σ entonces también son lenguajes sobre Σ:

1. Unión, $\mathbf{A} \cup \mathbf{B}$
2. Intersección, $\mathbf{A} \cap \mathbf{B}$
3. Diferencia, $\mathbf{A} - \mathbf{B}$
4. Complemento, $\bar{\mathbf{A}} = \Sigma^* - \mathbf{A}$

1.1.6. Clausura de Kleene de un Lenguaje

La potencia de un lenguaje L sobre Σ y $n \in \mathbb{N}$, donde L^n es el conjunto de todas las concatenaciones de n cadenas de L, de todas las formas posibles.[3].

Definición 1.1.6.1 (Potencias de un Lenguaje)**.** Sea L^n la potencia de un lenguaje de la siguiente forma.

1. $L^0 = \{\lambda\}$,
2. $L^{n+1} = L^n L$.

La clausura de Kleene es la unión de todas las potencias de L y se denota L^*, donde L^* consta de todas las concatenaciones de cadenas de L, incluyendo λ.

Definición 1.1.6.2 (Clausura de Kleene)**.**

$$L^* = \bigcup_{i \geq 0} L^i = L^0 \cup L^1 \cup L^2 \cup ... \cup L^n ... \tag{1.1.8}$$

De esta manera se define la **clausura positiva** de un lenguaje L, $L \subseteq \Sigma^*$, y se denota por L^+.

Definición 1.1.6.3 (Clausura Positiva)**.**

$$L^+ = \bigcup_{i \geq 1} L^i = L^1 \cup L^2 \cup ... \cup L^n ... \tag{1.1.9}$$

Nótese que $L^* = L^+ \cup \{\lambda\}$ y que $L^* = L^+$ si y solamente si $\lambda \in L$.

1.2. Lenguajes Regulares

Dados los lenguajes básicos $\emptyset$, $\{\lambda\}$, $\{x\}$, $x \in \Sigma$, se define de forma recursiva un lenguaje regular de la siguiente manera

1. $\emptyset$, $\{\lambda\}$, $\{x\}$ son lenguajes regulares
2. Si A Y B son lenguajes regulares entonces tambien son lenguajes regulares:

$$A \cup B \tag{1.2.1}$$
$$A \circ B \tag{1.2.2}$$
$$A^* \tag{1.2.3}$$

Note que todo `lenguaje finito` L es `regular`, dado que L se puede obtener con uniones y concatenaciones y todo `lenguaje regular` pueden ser reconocido y aceptado por un autómata finito.

1.3. Expresiones Regulares

Definición 1.3.0.4 (Expresión Regular)**.** Dado un alfabeto Σ, una `expresión regular` sobre Σ es toda cadena que cumple la siguiente definición recursiva:

1. $\emptyset$ es una expresión regular que representa al lenguaje $\emptyset$.
2. λ es una expresión regular que representa el lenguaje $\{\lambda\}$.
3. x es una expresión regular que representa el lenguaje $\{x\}$, $x \in \Sigma$.
4. Si R y S son expresiones regulares sobre Σ, entonces también lo son sus operaciones:

$$(R \cup S) \tag{1.3.1}$$
$$(R) \circ (S) \tag{1.3.2}$$
$$(R)^* \tag{1.3.3}$$
$$\tag{1.3.4}$$

Una expresión regular puede tener la siguiente notación:

$$L(R) := \text{ lenguaje representado por } R \tag{1.3.5}$$

Es claro que todo lenguaje regular puede ser representado por una expresión regular, pero no necesariamente esta representación es única, puede exitir el caso en que diferentes expresiones regulares representen el mismo lenguaje.

Ejemplo 1. Expresión Regular Encontrar una expresión regular para el lenguaje de todas las cadenas que contienen un número par de $a$es.

1. $b^*(b^*ab^*ab^*)^*$.
2. $(ab^*a \cup b)^*$.

1.4. Gramáticas Regulares

Definición 1.4.0.5 (Gramáticas Regulares)**.** Una gramática regular esta definida por un cuádruplo $\langle T, N, S, R\rangle$ en donde:

1. T es un alfabeto (conjunto de símbolos terminales).
2. N es un conjunto (conjunto de símbolos no terminales).
3. S donde $S \in N$ es el símbolo inicial.
4. R es un conjunto finito de reglas.

Una gramática regular es lineal por la izquierda o por la derecha , las gramaticas parten de una variable llamada símbolo inicial y luego se aplican repetidamente las reglas gramaticales hasta que ya no haya variables en la cadena. En compiladores se le dicen terminales a los elementos del alfabeto T, y no terminales a las variables.

1.5. Autómata Finito

En esta sección se describen los autómatas finitos como modelos matemáticos que son tratados como máquinas abstractas debido a los computos que realizan dependiendo de la entrada que reciben estos producen una salida, en este caso se presenta como los autómatas finitos deterministas, autómatas finitos no deterministas y autómatas finitos con transiciones lambda procesan cadenas las cuales pueden ser aceptadas o rechazadas.

1.5.1. Autómata Finito Determinista

Los autómatas finitos deterministas son maquinas abstractas que procesan cadenas de entrada las cuales pueden ser aceptadas o rechazadas, el autómata consta de una unidad de control o unidad de memoria que tiene unas configuraciones internas llamadas estados finales o de aceptación y el estado inicial.

Definición 1.5.1.1 (Autómata finito determinista)**.** Un *autómata finito determinista* se define como una 5-tupla

$$M = \langle S, i, \Sigma, \delta, F \rangle \tag{1.5.1}$$

donde:

1. $S = e_0, e_1, ...e_n$ es un conjunto finito no vacío, denominado conjunto de estados.
2. Σ es un conjunto finito denominado alfabeto.
3. $\delta : S \times \Sigma \to S$: función de transición del autómata, a partir de un estado y un símbolo del alfabeto obtiene un nuevo estado.
4. $i \in S$ estado denominado inicial.
5. F: $F \subseteq S$, se denomina conjunto de estados de aceptación o finales.

Note que para que un autómata finito determinista sea bien definido debe cumplir con unas condiciones, dado el número de transiciones que salen de cada estado debe ser igual a la cantidad de caracteres del alfabeto, puesto que δ es una función que esta definida para todas las entadas posibles, debe tener exactamente un solo estado inicial, en cambio los estados finales pueden tener un máximo de $|S|$ o no tener ninguno, debe seguir las transiciones es decir:

Definición 1.5.1.2 (Transición de un Autómata)**.** Si un autómata finito determinista se encuentra en un estado e y recibe un caracter θ pasa al estado e' si y solo si $\delta(e, \theta) = e'$, esto es, si $((e, \theta), e') \in \delta$.[1]

Dado un autómata finito se denomina al lenguaje reconocido por este autómata al conjunto de palabras que al operar sobre el estado inicial dan como resultado un estado de aceptación.

Definición 1.5.1.3 (Lenguaje aceptado por un autómata)**.** Sea $M = \langle S, i, \Sigma, \delta, F \rangle$ un autómata finito. El *lenguaje aceptado* por M, $L(M) \subset \Sigma^*$ se define como:

$$L(M) := \{\alpha \in \Sigma^* : M \text{ termina el procesamiento de } \alpha \text{ en un estado } e \in F\}. \tag{1.5.2}$$

1.5.2. Autómata Finito No Determinista

Los autómatas finitos no deterministas son muy similares a los autómatas finitos deterministas a diferencia que para cada estado $e \in S$ y cada $\alpha \in \Sigma$, la transición $\delta(e, \alpha)$ puede consistir en más de un estado o puede no estar definida.[3]

Definición 1.5.2.1 (autómata finito no determinista)**.** Se define como la 5-tupla de la forma $M = \langle S, i, \Sigma, \delta, F \rangle$, donde:

1. S es un conjunto finito no vacío, cuyos elementos se denominan estados.
2. Σ es un alfabeto.
3. $i \in S$, se denomina estado inicial.
4. $F \subseteq S$, se denomina conjunto de estados de aceptación o finales. véase
5. $$\begin{aligned} &\delta : S \times \Sigma \longrightarrow \wp(S) \\ &(e, x \in \Sigma) \longmapsto \delta(e, x) = \{e_{i1}, e_{i2}, \ldots, e_{ik}\} \end{aligned}$$

   donde $\wp(S)$ es el conjunto de subconjuntos de S.

Los autómatas finitos no deterministas pueden procesar las cadenas de entrada de diferentes maneras donde $\alpha \in \Sigma^*$, es decir que pueden existir varias trayectorias desde el estado e_0, etiquetados con los símbolos de α. Teniendo en cuenta la definición 1.5.1.3, note que para que una cadena α sea aceptada, debe existir por lo menos un cómputo en el que α sea procesada completamente y finalice M en un estado de aceptación.

1.5.3. Equivalencia entre AFD y AFND

Los modelos de autómatas finitos deterministas y autómatas finitos no deterministas son computacionalmente equivalentes, se puede decir que los autómatas finitos deterministas (AFD) son un `subcojunto` propio de los no deterministas (AFND).

Teorema 1.5.3.1 (Equivalencia)**.** *Dado un AFND $M = \langle S, i, \Sigma, \delta, F \rangle$ se puede construir un AFD M'* ***equivalente a*** *M, tal que $L(M) = L(M')$. Ver demostración [3].*

Este teorema establece que el no-determinismo se puede eliminar, por lo tanto los autómatas finitos deterministas y los autómatas finitos no deterministas aceptan los mismos lenguajes.

1.5.4. Autómata Finito con Transiciones λ

Definición 1.5.4.1 (autómata finito con transiciones lambda)**.** Es un autómata no determinista $M = \langle S, i, \Sigma, \delta, F \rangle$, en el que su función de transición esta definida cómo:

$$\delta : S \times (\Sigma \cup \lambda) \longrightarrow \wp(S) \tag{1.5.3}$$

Los autómatas finitos que aceptan transiciones λ permiten al autómata cambiar internamente de estado sin procesar el símbolo leído. En los autómatas con transiciones λ pueden existir computos infinitos que se presentan cuando el autómata ingresa a un estado donde tiene varias transiciones λ encadenadas que retornan al mismo estado.

1.5.5. Equivalencia entre AFN-λ y AFND

Un autómata finito con transiones λ es computacionalmente equivalente a un autómata finito no determinista, dado que las transiciones λ pueden ser eliminadas para añadir transiciones que las simulen sin alterar el lenguaje aceptado.

Teorema 1.5.5.1. *Dado un Autómata Finito con Transiciones λ $M = \langle S, i, \Sigma, \delta, F\rangle$, se puede construir un Autómata Finito no Determinista M' equivalente a M, es decir, tal que $L(M) = L(M')$. Ver demostración [3].*

1.5.6. Expresiones regulares y Gramáticas Regulares

En secciones anteriores se han dado las descripciones de lenguajes regulares, expresiones regulares y autómatas finitos. Como se puede observar existe una equivalencia computacional entre los modelos $AFD \equiv AFND \equiv AFN - \lambda$, por lo tanto estos autómatas aceptan los mismos lenguajes los cuales son llamados `Lenguajes Regulares`.

Expresiones regulares

Teorema 1.5.6.1 (Teorema de Kleene)**.** *Un lenguaje es regular si y sólo si es aceptado por un autómata finito (AFD o $AFND$ o $AFN - \lambda$). Ver demostración [3].*

La demostración se compone de dos partes, la primera parte dice que dado un lenguaje regular L existe un AFN-λ tal que $L(M) = L$, la segunda parte muestra que a partir de un AFD M se puede encontrar una expresión regular R tal que $L(M) = R$.

Las expresiones regulares son fórmulas que representan un lenguaje, por ejemplo R"$\emptyset$", representa el conjunto vacío $\{\}$, a diferencia de los autómatas, en las expresiones regulares no existen algoritmos para comparar dos expresiones regulares, para poder realizar esta comparación se convierte la expresión regular a un AFD. La operación de la suma es conmutativa en expresiones regulares, entonces dadas dos espresiones regulares R y S, la equivalencia se basa directamente en la conmutatividad de la unión de conjuntos es decir $R + S = S + R$. [1]

A partir de la definición del teorema 1.5.6.1 se puede decir que, dado un lenguaje regular que es aceptado por un autómata finito se puede determinar una expresión regular que acepte el mismo lenguaje o viceversa dada la expresión determinar el autómata finito que acepte el lenguaje.

Definición Para todo AFND existe una expresión regular R tal que $L(M) = R$.

De esta manera al establecer la equivalencia entre las expresiones regulares y los autómatas finitos no deterministas, automáticamente queda establecida la equivalencia entre las expresiones regulares y los autómatas finitos deterministas.

Gramáticas Regulares

Teorema 1.5.6.2 (Gramáticas regulares)**.** *Un lenguaje es regular si y sólo si es generado por una gramática regular. Ver demostración [2].*

El lenguaje generado por una gramática $G = \langle T, N, S, R\rangle$ se define como

$$L(G) := \alpha \in \Sigma^* : S \xrightarrow{+} \alpha$$

Una gramática es regular por la derecha si todas las producciones tienen la siguiente forma

$$\begin{aligned} A &\longrightarrow wB, \\ A &\longrightarrow w \end{aligned}$$

donde A y B pertenecen a T y $w \in \Sigma^*$ y una gramática es regular por la izquierda si todas las producciones tienen la siguiente forma

$$\begin{aligned} A &\longrightarrow Bw, \\ A &\longrightarrow w \end{aligned}$$

donde A y B pertenecen a T y $w \in \Sigma^*$.

Partiendo del teorema 1.5.6.2 se construye un procedimiento para que a partir de una gramática regular, se pueda construir un autómata finito y viceversa. Este procedimiento asocia los símbolos no terminales con los estados de un autómata, de esta manera para cada regla $A \longrightarrow wB$, de la gramática, existe una transición en el autómata A, w, B. Para los casos $A \longrightarrow w$, la transición que se crea es $A \longrightarrow wZ$,, donde Z es un nuevo estado para el que no existe un no terminal asociado, Z es el único estado final del autómata.[1] Se debe tener en cuenta que si el autómata finito determinista acepta la cadena vacía, la gramática regular no es capaz de generarla, ya que ninguna gramática regular es capaz de generar λ, para estos casos basta con generar una gramática regular que acepta el mismo lenguaje que el del autómata finito determinista excepto por la cadena vacía.

1.5.7. Operaciones Regulares

En esta sección se expone como los lenguajes regulares son cerrados bajo las operaciones conjuntivas básicas, como la unión, intersección, complemento, concatenación, estas operaciones pemitiran construir autómatas mas complejos a partir de los mas sencillos.

Teorema 1.5.7.1 (Unión de lenguajes regulares)**.** *Sean L_1 y L_2 lenguajes regulares. Entonces $L_1 \cup L_2$ es regular.*

Demostración 1.5.7.2. *Sean $M_1 = \langle S_1, i_1, \Sigma, \delta_1, F_1\rangle$ y $M_2 = \langle S_2, i_2, \Sigma, \delta_2, F_2\rangle$ automátas finitos deterministas conforme a la definición 1.5.1.1, tales que $L(M_1) = L_1$ y $L(M_2) = L_2$.*

Definimos $M = \langle S, i, \Sigma, \delta, F\rangle$ tal que:

1. $S = S_1 \times S_2$

2. $i = (i_1, i_2)$

3. $\delta((e, e'), x_{i+1}) = (\delta_1(e, x_{i+1}), \delta_2(e', x_{i+1}))$ donde $e \in S_1$y $e' \in S_2$

4. $F = (S_1 \times F_2) \cup (F_1 \times S_2)$

Veamos que $L(M) = L_1 \cup L_2$. Sea $\alpha = x_1x_2...x_n \in L(M)$. Como $\alpha \in L(M)$, existe una secuencia de estados

$$(e_0, e'_0), (e_1, e'_1), ..., (e_n, e'_n) \in S \tag{1.5.4}$$

tal que

$$(e_0, e'_0) = i, \quad (e_n, e'_n) \in F, \quad \delta((e_i, e'_i), x_{i+1}) = (e_{i+1}, e'_{i+1}).$$

Nótese que $e_0, e_1, \ldots, e_n \in S_1$, que $e'_0, e'_1, \ldots, e'_n \in S_2$ y por definición de δ se tiene que:

$$e_0 = i_1 \text{ y } e'_0 = i_2, \quad (e_n, e'_n) \in F = (S_1 \times F_2) \cup (F_1 \times S_2),$$
$$\delta(e_i, x_{i+1}) = e_{i+1}, \quad \delta(e'_i, x_{i+1}) = e'_{i+1}.$$

Dado que $(e_n, e'_n) \in (S_1 \times F_2) \cup (F_1 \times S_2)$, entonces $(e_n, e'_n) \in (S_1 \times F_2)$ o $(e_n, e'_n) \in (F_1 \times S_2)$ sin embargo sea cual sea el caso siempre $e_n \in F_1$ o $e'_n \in F_2$ por tanto $\alpha \in L_1$ o $\alpha \in L_2$ lo que significa que

$$\alpha \in L_1 \cup L_2$$

Sea $\alpha \in L_1 \cup L_2$, existen estados $e_0, e_1, ..., e_n \in S_1$ tal que

$$e_0 = i_1, \quad e_n \in F_1, \quad \delta_1(e_i, x_{i+1}) = e_{i+1}.$$

De esta manera

$$\alpha \in L(M_1) = L_1$$

Teorema 1.5.7.3 (Intersección de lenguajes regulares)**.** *Sean L_1 y L_2 lenguajes regulares entonces $L_1 \cap L_2$ es regular.*

Demostración 1.5.7.4. *Sean $M_1 = \langle S_1, i_1, \Sigma, \delta_1, F_1\rangle$ y $M_2 = \langle S_2, i_2, \Sigma, \delta_2, F_2\rangle$ automátas finitos deterministas conforme a la definición 1.5.1.1, tales que $L(M_1) = L_1$ y $L(M_2) = L_2$.*

Definimos $M = \langle S, i, \Sigma, \delta, F\rangle$ tal que:

1. $S = S_1 \times S_2$

2. $i = (i_1, i_2)$

3. $\delta((e, e'), x_{i+1}) = (\delta_1(e, x_{i+1}), \delta_2(e', x_{i+1}))$ con $e \in S_1$ y $e' \in S_2$

4. $F = F_1 \times F_2$

Probaremos que

$$L(M) = L_1 \cap L_2.$$

Sea la cadena $\alpha = x_1x_2...x_n \in L(M)$, como $\alpha \in L(M)$, existe una secuencia de estados

$$(e_0, e'_0), (e_1, e'_1), ..., (e_n, e'_n) \in S$$

tal que

$$(e_0, e'_0) = i, \quad (e_n, e'_n) \in F, \quad \delta((e_i, e'_i), x_{i+1}) = (e_{i+1}, e'_{i+1}).$$

Note que $e_0, e_1, ..., e_n \in S_1$ y por definición de δ se tiene que:

$$e_0 = i_1, \quad e_n \in F_1, \quad \delta(e_i, x_{i+1}) = e_{i+1}.$$

es decir,

$$\alpha \in L(M_1) = L_1.$$

De manera análoga,

$$\alpha \in L(M_2) = L_2$$

Por tanto

$$\alpha \in L_1 \cap L_2$$

Sea $\alpha \in L_1 \cap L2$, existe estados

$$e_0, e_1, ..., e_n \in S_1$$

tal que

$$e_0 = i_1, \quad e_n \in F_1, \quad \delta_1(e_i, x_{i+1}) = e_{i+1}.$$

Además existe

$$e'_0, e'_1, ..., e'_n \in S_2$$

tal que

$$e'_0 = i_2, \quad e'_n \in F_2, \quad \delta_2(e'_i, x_{i+1}) = e'_{i+1}.$$

De esta manera

$$(e_0, e'_0) = (i_1, i_2) = i, \quad (e_n, e'_n) \in F_1 \times F_2, \text{ y}$$
$$(\delta_1(e_i, x_{i+1}), \delta_2(e'_i, x_{i+1})) = (e_{i+1}, e'_{i+1}).$$

Por tanto

$$\alpha \in L(M)$$

Teorema 1.5.7.5 (Concatenación de lenguajes regulares)**.** *Sean L_1 y L_2 lenguajes regulares entonces $L_1 \circ L_2$ es regular.*

Similar a las ideas anteriores en la construcción de las demostraciones de los teoremas de unión e intersección, la idea bajo la demostración de este teorema es tomar los dos AFND M_1 y M_2 que aceptan los lenguajes L_1 y L_2 respectivamente y combinarlos para formar otro AFND M que acepte la concatenación de estos lenguajes de la siguiente manera:

1. El conjunto de estados de M será la unión de los conjuntos de estados de M_1 y M_2.
2. Se deben conservar las mismas transiciones definidas por δ_1 y δ_2.
3. Asignamos como estado inicial de M al estado inicial de M_1.
4. Por cada estado de aceptación de M_1 se crea una transición λ dirigida al estado inicial de M_2.
5. El conjunto de estados de aceptación de M será el conjunto de estados de aceptación de M_2.

Demostración 1.5.7.6. *Sean $M_1 = \langle S_1, q_1, \Sigma, \delta_1, F_1\rangle$ y $M_1 = \langle S_2, q_2, \Sigma, \delta_2, F_2\rangle$ autómatas finitos, tales que $L(M_1) = L_1$ y $L(M_2) = L_2$. Ver demostración [5].*

Construya $M = \langle S, q_1, \Sigma, \delta, F_2\rangle$ tal que $L(M) = L_1 \circ L_2$.

1. $S = S_1 \cup S_2$.
2. El estado q_1 de M es el mismo estado inicial de M_1.
3. Los estados de aceptación F_2 son los mismos estados de aceptación de M_2.
4. Se define δ para cada $q \in S$ y cualquier $a \in \Sigma_\epsilon$ de la siguiente manera:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in S_1 \text{ y } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ y } a \neq \lambda \\ \delta_1(q,a) \cup q_2 & q \in F_1 \text{ y } a = \lambda \\ \delta_2(q,a) & q \in S_2 \end{cases}$$

Teorema 1.5.7.7 (Complemento de Lenguajes Regulares). *Sea L un lenguaje regular entonces $\overline{L}$ es regular.*

Demostración 1.5.7.8. *Sea $M = \langle S, i, \Sigma, \delta, F\rangle$ un autómata finito determinista, tal que $\overline{M} = \langle S, i, \Sigma, \delta, S - F\rangle$.*

Definimos $M = \langle S, i, \Sigma, \delta, F\rangle$ un autómata finito determinista que acepta un lenguaje regular L, el autómata complementario $\overline{M}$ es aquel que acepta el lenguaje complemento de L, es decir, $\Sigma^* - L$, para esto hay que intercambiar los estados finales de M en no finales y viceversa.

1.5.8. Conversión de AFND a AFD

Dado $M = \langle S, i, \Sigma, \delta, F\rangle$ un autómata finito determinista

1.5.9. Minimización de estados

Dado $M = \langle S, i, \Sigma, \delta, F\rangle$ un autómata finito determinista, el siguiente algoritmo tiene como resultado un autómata equivalente minimal.

1. Remueva todos los estados que son inalcanzables desde el estado inicial.
2. Construya un grafo no dirigido G, cuyos vértices son los mismos estados de M.
3. Por cada par de estados p , q de M en donde alguno de los dos sea de aceptación, pero no ambos, cree una arista no dirigida (p, q).
4. Mientras aristas sean agregadas por el siguiente procedimiento repita:
5. Por cada pareja de estados distintos p y q de M y cada $x \in \Sigma$: agregue la arista (p, q) a G si la arista $(\delta(p, x), \delta(q, x))$ pertenece a G.
6. Por cada estado q, suponga $[q]$ es la colección de estados $r \in S$ tal que la arista (r, q) no esta en G.
7. Se construye el nuevo autómata finito determinista $M' = \langle S', i', \Sigma, \delta', F'\rangle$, donde $S' = \{[q] \in S\}$

   $\delta'([q], x) = [\delta(q, x)]$ para cada $q \in S$ y para cada $x \in \Sigma$.

   $q'_0 = [q_0]$ y

   $F' = \{[q] | q \in F\}$

1.5.10. Equivalencia

Definición 1.5.10.1 (Equivalencia)**.** Dos autómatas M_1 y M_2 son equivalentes, $M_1 \approx M_2$, cuando reconocen exactamente el mismo lenguaje, entonces si y sólo si los autómatas mínimos correspondientes de M_1 y M_2 son el mismo autómata.[5]

Existe una forma para decidir si dos autómatas finitos son equivalentes que no utiliza la propiedad de unicidad del autómata mínimo.
Dados dos autómatas finitos M_1 y M_2, que reconocen los lenguajes L_1 y L_2 respectivamente. Entonces el lenguaje L_3 es definido a partir de: $L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$, luego se cumple la siguiente equivalencia $L_1 = L_2 \Leftrightarrow L_3 = \emptyset$. Note que las operaciones de unión, intersección, y complemento son constructibles entonces se puede encontrar un Autómata Finito Determinista M_3 que reconoce el lenguaje L_3. Por lo tanto los autómatas M_1 y M_2 son equivalentes si y solo si el autómata M_3 no acepta ninguna palabra. Finalmente se dice que esta propiedad es decidible, ya que es equivalente a verificar que todos los estados accesibles de M_3 son de no aceptación.[5]

Otro forma para decidir si dos autómatas finitos son equivalentes es con el método del `Teorema de Moore`, este teorema dice que existe un algoritmo para decidir si dos autómatas finitos son equivalentes o no.
El algoritmo que mencionan en el `Teorema de Moore` consiste en la construcción de un árbol de comparación de autómatas que convierte el problema de la comparación de los lenguajes aceptados en un problema de comparación de estados de los autómatas.

Definición 1.5.10.2. Decimos que dos estados e y e' son compatibles si ambos son finales o ninguno de los dos es final. En caso contrario, son estados incompatibles.

Este algoritmo de comparación entre AFD_1 y un AFD_2 busca alguna secuencia de caracteres w que al realizar el recorrido simultáneamente se llega a estados incompatibles. Si no se encuentra esta secuencia, entonces los autómatas son equivalentes. Con este método se debe garantizar que fueron exploradas todas las posibles cadenas de caracteres w, esto puede ser un problema debido a que dicha búsqueda puede llegar a ser infinita, es por esto que se buscan todas las posibles combinaciones de estados mediante un árbol, este árbol de comparación se construye de la siguiente manera, dados dos autómatas $M = \langle S, i, \Sigma, \delta, F\rangle$ y $M = \langle S', i', \Sigma', \delta', F'\rangle$:

1. Inicialmente la raíz del árbol es el par ordenado (i, i') que contiene los estados iniciales de M y M' respectivamente.
2. Si en el árbol hay un par (r, r'), para cada caracter Σ se añaden como hijos suyos los pares (r_σ, r'_σ) donde $r_\sigma = \delta(r, \sigma)$, $r'_\sigma = (r', \sigma)$, si no están ya.
3. Si aparece en el árbol un par (r, r') de estados incompatibles, se interrumpe la construcción del mismo, concluyendo que los dos autómatas no son equivalentes. En caso contrario se continúa a partir del paso 2.
4. Si no aparecen nuevos pares (r_σ, r'_σ) que no estén ya en el árbol,se termina el proceso, concluyendo que los dos autómatas son equivalentes.

La prueba de este algoritmo de decisión la podemos encontrar en el libro [1].

CAPÍTULO 2

SiGrAλe

SiGrAλe es una herramienta gráfica interactiva desarrollada en el lenguaje de programación Java que permite el diseño de autómatas finitos deterministas, autómatas finitos no deterministas y autómatas finitos no deterministas con transiciones lambda. La herramienta soporta la evaluación de las cadenas por parte de los autómatas (decidir si una cadena hace parte o no de un lenguaje), operaciones de clausura entre autómatas como: unión, intersección, complemento, concatenación y clausura. Ademas incluye implementaciones de los algoritmos de conversión entre autómatas AFND a AFD, minimización de estados, equivalencia entre autómatas y representación de un autómata como expresión regular y viceversa.

2.1. Requerimientos

La herramienta SiGrAλe cumple el siguiente paquete de requerimientos:

2.1.1. Requerimientos funcionales

La herramienta SiGrAλe, permite:

1. Crear un autómata a partir del diseño de su diagrama de estados.
2. Guardar un autómata en un archivo de tal manera que el usuario pueda visualizarlo posteriormente.
3. Guardar el diagrama de estados de un autómata como una imagen.
4. Determinar si una cadena pertenece al lenguaje aceptado por un autómata en particular.
5. Llevar a cabo operaciones con autómatas: unión, intersección, concatenación, y complemento.
6. Realizar la conversión de autómatas finitos no deterministas a deterministas.

7. Realizar minimización de autómatas.
8. Determinar la equivalencia entre dos autómatas.
9. Encontrar una expresión regular equivalente a un autómata dado.
10. Encontrar un autómata equivalente a un expresión regular dada.
11. Encontrar una gramática regular equivalente a un autómata dado.
12. Encontrar un autómata equivalente a un gramática regular dada.

2.1.2. Requerimientos no funcionales

1. El panel de diseño de diagramas de autómatas es sencillo de usar.
2. La herramienta `SiGrAλe` fue diseñada de tal manera que su código puede ser mantenible.
3. La herramienta `SiGrAλe` es escalable ya que permite que a futuro se puedan desarrollar nuevas funcionalidades.

2.2. Arquitectura de `SiGrAλe`

La herramienta `SiGrAλe` fue diseñada de tal manera que separa la implementación de los autómatas del conjunto de operaciones (algoritmos) que se puede realizar con ellos mediante un diseño basado en la definición de interfaces.

Esto permite que el conjunto de clases pueda ser tratado como una librería.

2.2.1. Clases y estructuras de datos

- Clase `Simbolo` representa un símbolo.
- Clase `Alfabeto` representa un alfabeto.
- Interfaz `Estado` define un conjunto de métodos que debe proporcionar cualquier clase que desee ser tratada como un estado de un autómata. En particular cualquier clase `Estado` basta con implementar el método `getEtiqueta` que retorna una etiqueta que es una representación única del estado, además de ser la respectiva etiqueta que será visualizada en el diagrama de transiciones del autómata.

  Además de lo anterior debe implementar la interfaz `Comparable<Estado>` y por ende el método `compareTo` el cual permitirá a la herramienta distinguir los elementos en el conjunto de estados.

  La clase `EstadoImpl` es la implementación más sencilla de `Estado` en `SiGrAλe`.

  La definición simple de esta interfaz tiene la ventaja que permite en los algoritmos

poder utilizar otra implementación de la clase Estado que implemente otras funciones o este compuesta de otros elementos sin alterar la naturaleza del autómata.

- Interfaz `Transicion` define un conjunto de métodos que debe proporcionar cualquier clase que desee ser tratada como una transición de un autómata.

- Interfaz `Automata` define un conjunto de métodos que debe proporcionar cualquier clase que desee ser tratada como un autómata. La definición de esta clase permite, cómo se menciona en el capítulo 3, que el visualizador de autómatas producto del desarrollo de la herramienta pueda ser utilizado de manera independiente a `SiGrAλe`.

  Las clases `AutomataFinito` y `AutomataFinitoImpl` son la implementación de la Interfaz `Automata` en `SiGrAλe`.

- Interfaz `Automatas` define métodos que representan el conjunto de operaciones regulares: unión, intersección, concatenación, complemento, clausura que se pueden llevar a cabo sobre Autómatas. Además métodos como equivalencia, minimización y conversión de autómatas a deterministas. Por medio de la interfaz `Automata` fue posible lograr la separación de la implementación de estos de métodos de la implementación del autómata en si, así como de la visualización del mismo.

  La clase `AutomatasImpl` es la implementación de la interfaz `Automatas` en `SiGrAλe`.

- Clase abstracta `AutomataFinito` implementa la clase `Automata` y ofrece implementación de algunos de los métodos ayudantes que están definidos en la clase `Automata`.

  Llamamos *métodos ayudantes* en una clase `A` a los métodos que nos permiten acortar la escritura de código por que hacen puente entre métodos que se pueden acceder únicamente por medio de atributos u objetos obtenidos mediante la invocación métodos definidos en la clase `A`, por ejemplo, en este caso el método `agregarEstado` definido en la clase `Automata` es un método ayudante porque se encarga de invocar el método `getEstados` que retorna el conjunto de estados de un autómata, definido también en `Automata`, seguido del método `add` que agrega un estado a un autómata. Hay que notar que ambos métodos enunciados anteriormente `getEstados` y `add` no dependen de la implementación que se haga de `Automata` por tanto esta clase abstracta (`AutomatasImpl`) define varios de estos métodos que reducen el esfuerzo necesarios para la implementación de la interfaz `Automata`.

  Gracias a que la clase abstracta `AutomataFinito` reduce el esfuerzo de la implementación de varios de los métodos definidos en la interfaz `Automata`, esta puede ser extendida con el propósito de implementar la clase `Automata` y poder de esta manera llevar a cabo operaciones sobre autómatas y/o la visualización de estos.

- Clase `AutomataFinitoImpl` extiende la clase `AutomataFinito` y por tanto es también una implementación de la interfaz `Automata`. Esta clase representa el conjunto de estados de un autómata como un objeto de tipo `java.util.TreeSet` el cual es una implementación de un `java.util.Set` en `Java` que garantiza una complejidad temporal de $\log n$ para las operaciones básicas inserción, eliminación y consulta y por tanto es una muy buena opción a la hora de representar conjuntos en el lenguaje `Java`.

- Interfaz `FuncionDeTransicion` define un conjunto de métodos de tal manera que una clase que la implemente represente la función de transición de una autómata. Una clase que implemente esta interfaz es la encargada de almacenar la información necesaria sobre las transiciones del autómata así como de indicar dado un estado y un símbolo cual es la transición correspondiente. La implementación de esta clase no discrimina si el autómata es o no determinista y su estructura esta diseñada para soportar ambos autómatas transparentemente.

- Clase `FuncionDeTransicionImpl` es la implementación de la interfaz `FuncionDeTransicion` modela el conjunto de transiciones de un autómata utilizando un objeto de tipo `java.util.TreeMap` el cual es una implementación de un mapa `java.util.Map` basado en la implementación de un Árbol Rojinegro en `Java`, esta implementación de mapa garantiza una complejidad temporal de *logn* en las operaciones básicas como inserción, eliminación y consulta y por tanto es una muy buena opción a la hora de representar el conjunto de transiciones agrupado por pareja de estados.

2.2.2. Diagrama de clases

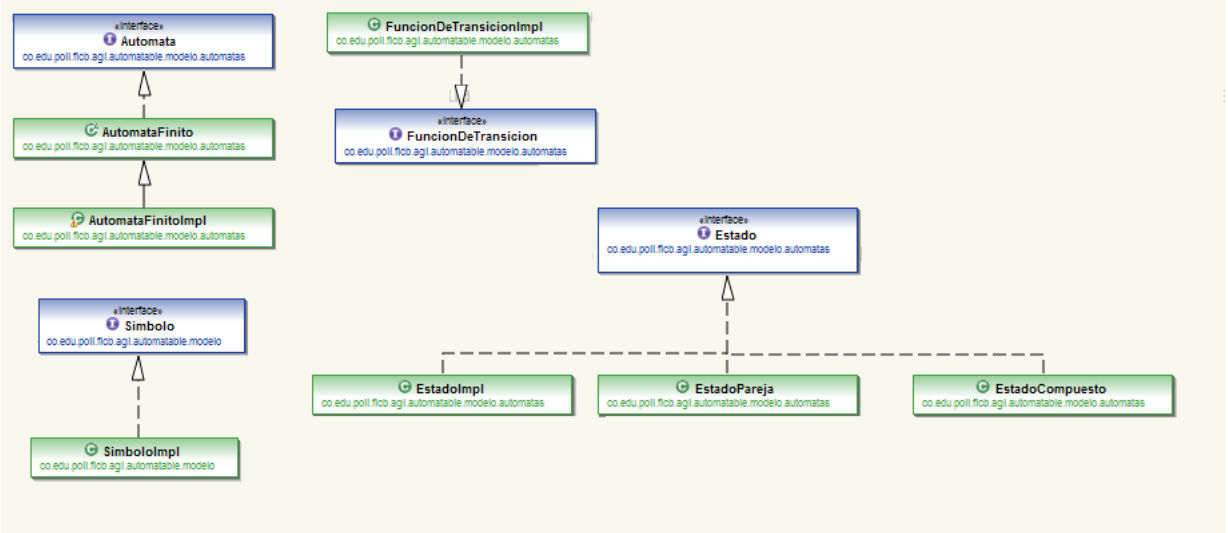

Figura 2.1: Diagrama de Clases de `SiGrAλe`

CAPÍTULO 3

Resultados y Conclusiones

3.1. Resultados

La herramienta `SiGrAλe` es un programa de escritorio desarrollado en lenguaje de programación `Java` que permite el diseño de autómatas y la comprobación de resultados en la teoría de autómatas y lenguajes, está diseñada con el propósito de apoyar los cursos de lenguajes formales, teoría de autómatas, teoría de la computación y afines.

La herramienta permite el diseño de autómatas finitos deterministas (AFD), autómatas finitos no deterministas (AFND) y autómatas finitos no deterministas con transiciones λ (AFN-λ), brindando una interfaz gráfica fácil de usar en la que el usuario puede definir cada uno de los componentes del diagrama de transiciones de un autómata como estados y transiciones, además de permitir la personalización de etiquetas, ubicación de estados arbitrariamente en el panel o ubicación automática de estados a partir de layouts predefinidos.

`SiGrAλe` proporciona al usuario la opción de convertir una AFND a un AFD. El algoritmo de conversión de un autómata finito no determinista a determinista tiene una complejidad temporal y espacial bastante alta sin embargo para nuestro propósito educativo el algoritmo se ejecuta en un tiempo discretamente pequeño siempre y cuando el autómata no sea muy grande, es un hecho que los autómatas a diseñar no sean tan complejos ni grandes. En cuanto a comprobación de resultados en teoría de autómatas la herramienta permite llevar a cabo operaciones binarias, unión, intersección y concatenación, y unaria como complemento entre autómatas realizando una ligera simulación de como procede el algoritmo en cuestión. También ofrece la posibilidad de encontrar una expresión regular equivalente a un autómata y viceversa. Ademas permite que a partir de una gramática regular se genere el autómata finito correspondiente a esta gramática o viceversa.

Las estructuras de datos utilizadas que representan conjuntos fueron escogidas de tal manera que la representación en java es semejante a la representación matemática de un autómata.

Consta de un panel que realiza el seguimiento de la ejecución del autómata, permitiendo observar el paso a paso del procesamiento de la cadena, destacando el estado actual y los

estados anteriores, arrojando como resultado la aceptación o no de la cadena por este autómata.

Los algoritmos clásicos de minimización de estados y equivalencia de autómatas también están incluidos y se convierten en una gran herramienta para el análisis del diseño de autómatas.

Cada uno de los autómatas diseñados en la herramienta `SiGrAλe` pueden ser salvados a archivos *.sgr para uso posterior, esto es, que puedan ser visualizados en otro pc con la misma herramienta. Además sus diagramas de transiciones pueden ser guardados como imágenes.

El diseño de la herramienta `SiGrAλe` fue construido de tal manera que sea sencilla la implementación en un futuro de nuevos desarrollos que aumenten la funcionalidad de `SiGrAλe` y no se convierta en un problema por las estructuras de datos utilizadas o porque simplemente no es flexible a la hora de agregar nuevos módulos, para esto se separo la implementación de los autómatas y el conjunto de operaciones (algoritmos) que se puede realizar con ellos mediante un diseño basado en la definición de interfaces y además se tienen algunos componentes reutilizables como el panel que visualiza los autómatas. El framework de visualización de grafos `JUNG` proporciona un conjunto limitado de `Shape`'s que son empleadas para dibujar las aristas de un grafo.

El código fuente de la herramienta puede ser encontrado en la siguiente dirección http://code.google.com/p/automatable/

3.1.1. Funcionalidad

Los algoritmos de las operaciones con autómatas se encuentran en la clase `AutomatasImpl`, la cual es una implementación de la interfaz `Automatas`.

La interfaz `Automatas` define los siguientes métodos, note que en cada uno de ellos no se hace distinción entre AFD, AFND, AFN-λ.

- `public Automata union(Automata m1, Automata m2)` este método recibe como parámetros dos autómatas y lleva a cabo el algoritmo subyacente bajo la demostración del teorema de unión de lenguajes regulares 1.5.7.1 retornando un autómata que acepta el lenguaje resultante de la unión de los lenguajes aceptados por los autómatas `m1` y `m2`.

- `public Automata interseccion(Automata m1, Automata m2)` este método recibe como parámetros dos autómatas y lleva a cabo el algoritmo subyacente bajo la demostración del teorema de intersección de lenguajes regulares 1.5.7.1 retornando un autómata que acepta el lenguaje resultante de la intersección de los lenguajes aceptados por los autómatas `m1` y `m2`.

- `public Automata concatenacion(Automata m1, Automata m2)` este método recibe como parámetros dos autómatas y lleva a cabo el algoritmo subyacente bajo la demostración del teorema de concatenación de lenguajes regulares 1.5.7.5 retornando un autómata que acepta el lenguaje resultante de la concatenación de los lenguajes aceptados por los autómatas `m1` y `m2`.

- `public Automata clausura(Automata m1)` este método recibe como parámetro un autómata y lleva a cabo el algoritmo subyacente bajo la demostración del teorema de clausura de lenguajes regulares retornando un autómata que acepta el lenguaje resultante de la clausura del lenguaje aceptado por el autómata `m`.

- `public Automata complemento(Automata m)` este método recibe como parámetro un autómata y lleva a cabo el algoritmo subyacente bajo la demostración del teorema de complemento de lenguajes regulares 1.5.7.5 retornando un autómata que acepta el lenguaje complemento de lenguaje aceptado por el autómata `m`.

- `public Automata minimizar(Automata m)` este método recibe como parámetro un autómata y lleva a cabo el algoritmo de minimización de estados **??** retornando un autómata equivalente al autómata `m` pero con un número mínimo de estados.

- `public ExpresionRegular convertirAER(Automata m)` este método recibe como parámetro un autómata y lleva a cabo el algoritmo de conversión de autómatas a expresión regular retornando una expresión regular que representa el mismo lenguaje que acepta el autómata `m`.

- `public Automata convertirAAutomata(ExpresionRegular er)` este método recibe como parámetro una expresión regular y lleva a cabo el algoritmo de conversión de expresión regular a autómata retornando un autómata que acepta el mismo lenguaje que representa la expresión regular `er`.

- `public boolean sonEquivalentes(Automata m1, Automata m2)` este método recibe como parámetros dos autómatas y lleva a cabo el algoritmo de equivalencia de autómatas retornando si los dos autómatas son equivalentes es decir si aceptan el mismo lenguaje.

- `public boolean convertirADeterminista(AutomataFinitoNoDeterminista m)` este método recibe como parámetro un autómata finito no determinista `m1` y retorna un autómata finito determinista equivalente llevando a cabo el algoritmo de conversión de autómata finito no determinista a determinista.

3.1.2. Interfaz de usuario

`SiGrAλe` está desarrollada utilizando los componentes `Swing` para la construcción de la interfaz gráfica. Aunque principalmente funciona como una aplicación de escritorio (standalone) es posible utilizarla en un ambiente WEB incluyéndola como `Applet` en una página.

Uno de los problemas a resolver durante el desarrollo de `SiGrAλe` fue el de la visualización de los diagramas de transición para los autómatas. El problema de mayor dificultad era la visualización de los autómatas que no hubiesen sido diseñados directamente utilizando el panel graficador, como en el caso en que se realizan operaciones entre autómatas debido a que el autómata resultante de la operación debía ser visualizado arbitrariamente y la responsabilidad de ello tenía que ser asumida por la herramienta. Por tanto se deseaba que la visualización de los autómatas que no eran diseñados por medio del panel, se realizara de forma inteligente y no al azar de tal manera que el grafo del autómata tuviera el mejor aspecto posible. Es por esto que se decidió buscar un framework que ofreciera diferentes

algoritmos de visualización de grafos, que fuera 100 % compatible con `Java` y se integrara fácilmente a la herramienta.

El framework de visualización y modelamiento de grafos `JUNG` fue la elección para visualizar los diagramas de transición de los autómatas AFN, AFND y AFN-λ en `SiGrAλe`. Este framework desarrollado en `Java` contiene varios algoritmos de visualización de grafos.

`JUNG` ofrece varias clases `Layout` que implementan diferentes algoritmos de visualización de grafos, entre las más relevantes se encuentran las siguientes:.

- `CircleLayout` implementa un algoritmo de visualización de grafos cuyo propósito es ubicar los vértices sobre un círculo y de manera que estos estén separados a una misma distancia uno del otro.
- `FRLayout` y `FRLayout2` implementan el algoritmo de visualización de grafos de forzamiento directo, cuyo propósito es posicionar los vértices del grafo de tal manera que todas las aristas tengan una distancia similar y halla el menor número de cruces posibles entre ellas. Ver [4].

La interfaz de usuario de `SiGrAλe` permite el diseño de autómatas finitos deterministas y no deterministas mediante la generación de su diagrama de transiciones. Para ello dispone de una ventana que consta de un menú, una barra de herramientas y un panel que visualiza el diagrama de un autómata.

la figura 3.1 muestra un ejemplo de la ventana de diseño de autómatas. Por medio de esta ventana es posible:

1. Elegir el modo de visualización del panel, sea este creación o edición de estados y transiciones.
2. Crear una estado del autómata haciendo click sobre cualquier parte del panel.
3. Editar las etiquetas de los estados.
4. Editar los símbolos de las transiciones.
5. Desplazar el autómata dentro del panel de visualización.
6. Aplicar un layout de `JUNG` al diagrama de estados.
7. Convertir el autómata diseñado a un autómata finito determinista.
8. Modificar la ubicación de estados dentro del panel.

En el menú se encuentra las siguientes opciones:

1. Guardar el autómata. Este archivo es salvado con una extensión `.sgr` propia de `SiGrAλe` es un archivo plano en formato `XML` que almacena el diseño para poder hacer uso posterior del mismo en la herramienta.
2. Aplicar los diferentes layouts de `JUNG`. Los layouts posibles son: KKLayout, FRLayout, CircleLayout, SpringLayout, Springlayout2, ISOMLayout.

3. Ejecutar autómata. Esta opción permite llevar a cabo la ejecución de una cadena en el autómata de tal manera que si ejecución puede ser visualizada de manera automática, paso a paso, o directa.

4. Cerrar la ventana.

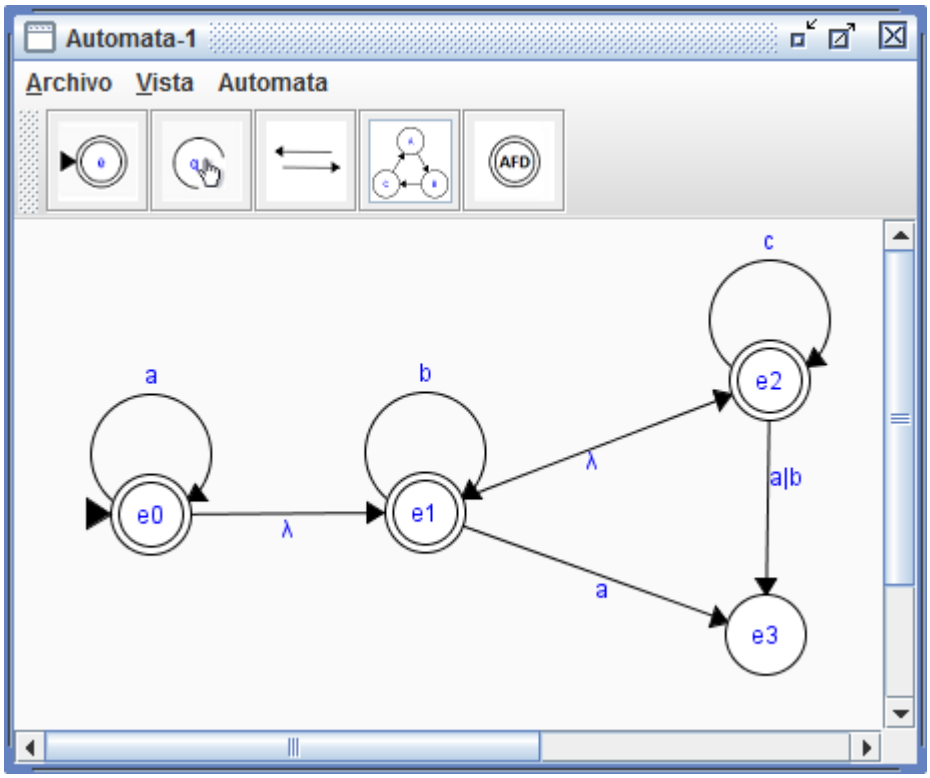

Figura 3.1: Diagrama de transiciones de un autómata finito no determinista que acepta el lenguaje representado por la expresión regular $a^*b^*c^*$.

Para llevar a cabo las operaciones entre autómatas `SiGrAλe` consta de dos ventanas en las cuales se puede visualizar los diagramas de transiciones de los autómatas o autómata involucrados en la operación a realizar y el autómata resultado de la operación.

La primera ventana 3.2 permite realizar operaciones unarias, sobre cualquier autómata finito tales como:

1. Complemento.

2. Clausura o estrella de Kleene.

Aunque no son operaciones entre autómatas esta ventana también permite llevar a cabo la conversión de autómata finto no determinista a finito determinista y permite realizar la minimización de estados a un autómata.

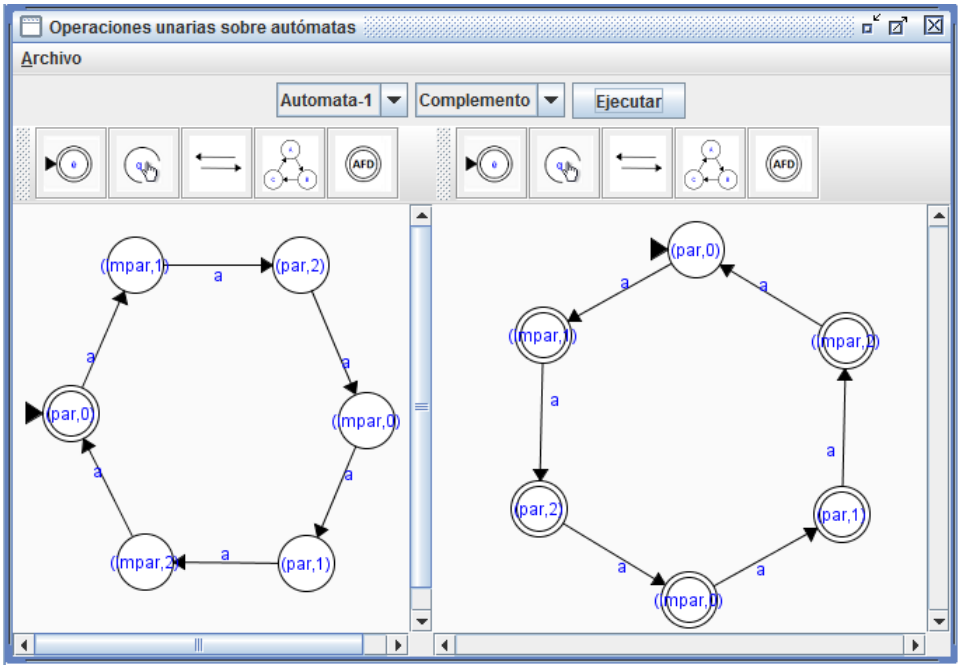

Figura 3.2: Ventana de operaciones unarias visualizando un autómata (izquierda) y calculando su respectivo complemento (derecha).

La segunda ventana 3.3 permite realizar operaciones binarias, sobre dos autómatas finitos tales como:

1. Unión.
2. Intersección.
3. Concatenación.
4. Equivalencia

Una de las funcionalidades con las que cuenta la herramienta es la de poder ejecutar un autómata dada una cadena. Para ello presenta una ventana que visualiza la ejecución del autómata permitiendo ingresar cadenas y ejecutarlas de modo automático o paso a paso. El modo automático ejecuta cada paso en intervalos de un segundo y mostrando como resultado final, la aceptación o no aceptación de la cadena por parte del autómata. El modo de ejecución paso a paso permite que el usuario decida cuando ejecutar un paso en la ejecución.

3.1.3. Visor de Autómatas

Durante el diseño de la herramienta surgió la necesidad de crear un panel cuya responsabilidad es la visualización del diagrama de estados de un autómata.

`PanelAutomata` fue la solución a este problema. Este hereda de la clase `JPanel` de `Swing` y proporciona la funcionalidad de visualización y edición de un autómata un obje-

to de cualquier clase que implemente la interfaz `Autómata` por esta razón el panel puede ser utilizado de manera independiente a `SiGrAλe` para cualquier otro propósito, por ejemplo, podría ser utilizado para permitir la visualización de autómatas generados por otras herramientas y/o programas por supuesto desarrollados en el lenguaje de programación `Java`.

Como se menciona anteriormente el único requisito para poder utilizar el panel de visualización es que se le proporcione un objeto de tipo `Automata`, esto garantiza por medio de la definición de los métodos en la interfaz que el panel sea capaz de visualizar correctamente el autómata dado y por tanto separa la visualización de la implementación del autómata.

3.2. Conclusiones

Se implemento `SiGrAλe` como una herramienta que apoye la enseñanza del curso de autómatas, gramáticas y lenguajes del Politécnico Grancolombiano, esta herramienta permite a los alumnos poder realizar la comprobación de resultados en teoría de autómatas y lenguajes de las operaciones de unión, intersección, concatenación, complemento y clausura de Kleene. La herramienta proporciona al usuario la opción de convertir una AFND a un AFD, también permite al usuario generar un autómata finito a partir de una expresión regular. `SiGrAλe` implementa los algoritmos de minimización de estados y equivalencia entre autómatas. La herramienta cuenta con un panel de visualización para realizar la simulación donde se evalúa si una cadena es aceptada o rechazada por un autómata.

`SiGrAλe` es una herramienta gráfica con una interfaz de usuario fácil de usar desarrollada en el lenguaje de programación `Java` esta aplicación permite, entre otras cosas, poder ejecutarla en un ambiente local como aplicación de escritorio y además ejecutarla por medio de un browser como parte de una página WEB. sección

3.3. Trabajo futuro

Es deseable la construcción de algunos módulos para la herramienta `SiGrAλe` que incluyan funcionalidades que comprendan el diseño y análisis de autómatas tales como:

- Autómatas de Pila.
- Lema de Bombeo.
- Lenguajes y gramáticas independientes del contexto.

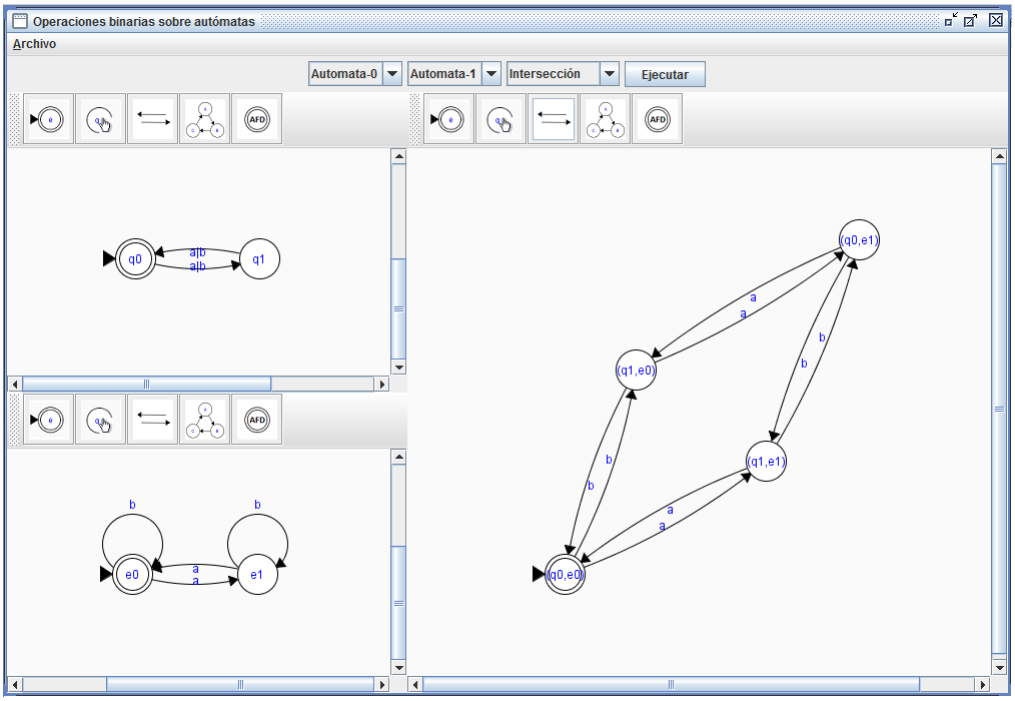

Figura 3.3: Ventana de operaciones binarias visualizando dos autómatas finitos (izquierda) y el autómata resultado de la operación calculada (derecha).

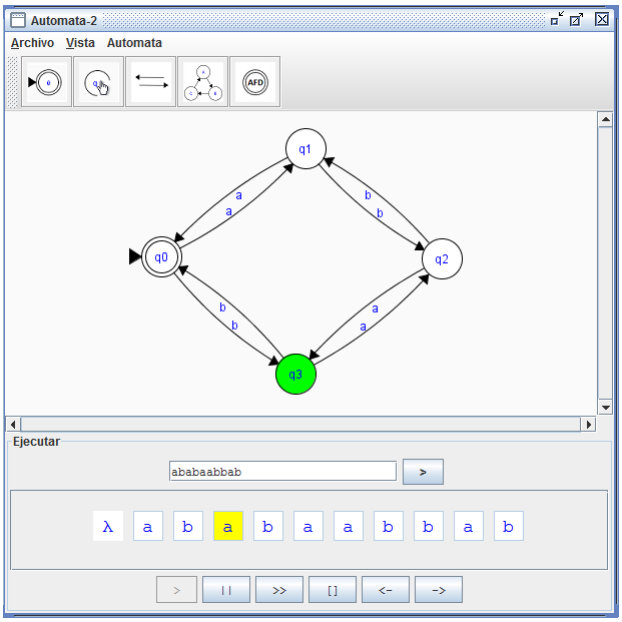

Figura 3.4: Ventana con simulación de la ejecuión del autómata con la cadena `abababbab`.

Bibliografía

[1] R. Brena. Autómatas y lenguajes, un enfoque de diseño.

[2] G. Brookshear. *Teoría de la Computación.* Addison Wesley Iberoamericana, 1993.

[3] R. D. Castro. *Teoría de la computación. Lenguajes, autómatas, gramáticas.* Universidad Nacional de Colombia, Bogotá D. C., 2004.

[4] Thomas M. J. Fruchterman, Edward, and Edward M. Reingold. Graph drawing by force-directed placement, 1991.

[5] L. M. Villodre R. C. Muñoz. *Lenguajes Gramáticas y autómatas.* Grupo editor ALFAOMEGA, 2002.

[6] M. Sipser. *Introduction to Theory of Computation.* Unibiblos, Universidad Nacional de Colombia, second edition edition, 2004.