Universidad Complutense de Madrid

Facultad de Informática

Trabajo de fin de grado en Ingeniería Informática

2020-21

COMPROBACIÓN DE EQUIVALENCIA ENTRE ESPECIFICACIONES BASADAS EN EXPRESIONES REGULARES

Checking equivalence of specifications based on regular expressions

Alumnos:
Raúl Benito Montoro
Xukai Chen

Director:
José Luis Sierra Rodríguez

Resumen

En este trabajo, realizamos un estudio sobre los diferentes algoritmos necesarios (y algunas alternativas) en la comprobación de equivalencia entre dos expresiones regulares, con todos los pasos que ello implica: creación de autómatas finitos, determinación de aquellos que lo necesiten y comprobación de la equivalencia.

El trabajo culmina con la creación y pruebas de una herramienta que admite dos especificaciones basadas en expresiones regulares y permite hacer comparaciones entre ellas, de manera total o parcial, utilizando los diferentes algoritmos de comprobación de la equivalencia entre expresiones regulares implementados.

Palabras clave: algoritmo Hopcroft-Karp, equivalencia, expresiones regulares, algoritmo de Thompson, algoritmo derivadas, algoritmo derivadas parciales, algoritmo Berry-Sethi, algoritmo seguidores, autómata finito, determinación de autómatas.

Abstract

In this work, we study the different algorithms needed (and some alternatives) for checking the equivalence of two regular expressions, with all the steps involved: creation of finite automata, determination of those that require it and equivalence checking between the resulting deterministic automata.

The work will finish with the creation and testing of a tool that can receive two regular expression-based specifications and makes it possible to compare them, partially or completely, with the help of the different algorithms for checking the equivalency of regular expressions.

Keywords: Hopcroft-Karp algorithm, equivalence, Thompson algorithm, Derivatives algorithm, Partial Derivatives algorithm, Berry-Sethi algorithm, followers algorithm, finite automata, automata determination.

Contenido

1. Introducción

1.1 – Motivación

Según la RAE[1]: "un patrón es un modelo que sirve de muestra para sacar otra cosa igual". Por ejemplo, en matemáticas, una secuencia de números que se dirige por una función; en genética, un conjunto de genes que provocan una característica distintiva en un ser vivo…

Su detección es un problema recurrente en informática a la hora de automatizar procesos, y esa detección puede describirse a alto nivel recurriendo al formalismo de las expresiones regulares.

Una expresión regular es una cadena de caracteres que se utiliza para la detección de patrones en cadenas de mayor longitud. La flexibilidad de estas expresiones regulares nos permite encontrar diversos patrones, más o menos complejos y de estructura conocida, en secuencias de caracteres de mayor tamaño.

Debido a la importancia de este tipo de especificaciones, en este trabajo crearemos una herramienta que facilite su aprendizaje: en ella, el usuario introducirá dos especificaciones basadas en expresiones regulares que describan el léxico de un lenguaje formal, y la herramienta hará las comprobaciones de equivalencia que detallaremos a lo largo de este trabajo, informando al usuario de si ambas especificaciones detallan los mismos patrones o no.

1.2 – Motivation

According to the RAE: "a pattern is a model that serves as a sample to get something same". For example, in mathematics, a number sequence modeled by a mathematical function; in genetics, a set of genes that cause a distinctive characteristic in a living being...

Its detection is a recurrent problem in computer science. When it comes to automating processes, this detection can be described at a high level by resorting to the formalism of regular expressions.

A regular expression is a string of characters used to detect patterns in longer strings. The flexibility of these regular expressions allows us to find out various known structure patterns, more or less complex, in longer character sequences.

Due to the importance of this type of specifications, in this project we will create a tool that improves user's learning to define specifications: within the tool, the user will insert two specifications based on regular expressions that describe a formal language's lexicon, and the tool will perform the equivalence checks that we will detail throughout this work, which report to the user whether both specifications describe the same patterns or not.

[1] https://dle.rae.es/patr%C3%B3n

1.3 – Objetivos

En este trabajo planteamos los siguientes objetivos:

1. Estudiar los algoritmos necesarios para realizar la comprobación de la equivalencia de especificaciones basadas en expresiones regulares.
2. Diseñar distintos métodos de comprobación de la equivalencia.
3. Implementar un sistema para la comprobación de la equivalencia de especificaciones basadas en definiciones regulares.
4. Realizar una evaluación preliminar del sistema.

1.4 – Objectives

In this project we set out the following objectives:

1. Study the algorithms required to carry out equivalence checking of specifications based on regular expressions.
2. Design different equivalence checking methods.
3. Implement a system for checking the equivalence of specifications based on regular expressions.
4. Perform a preliminary evaluation of the system.

1.5 – Plan de trabajo

A continuación, detallamos el plan de trabajo seguido para abordar los distintos objetivos planteados.

1.5.1 – Estudio de los algoritmos

Para abordar el primer objetivo, investigaremos los algoritmos que nos permiten comprobar la equivalencia entre distintas expresiones regulares. Para ello, la comprobación de la equivalencia se reducirá a comprobar la equivalencia entre autómatas finitos deterministas. A este respecto, hemos encontrado el algoritmo de Hopcroft-Karp [11].

Estos autómatas finitos deterministas pueden obtenerse a partir de las expresiones regulares mediante diversos algoritmos: Thompson [1], seguidores [3], derivadas [4], derivadas parciales [4] o Berry-Sethi [6]. Algunos de estos algoritmos solo obtienen autómatas finitos no deterministas, por lo que también será necesario estudiar un algoritmo de determinación por subconjuntos (en sus dos versiones: con y sin λ-transiciones) [9].

En esta fase, estudiamos todos los algoritmos que implementamos en la herramienta.

1.5.2 – Diseño de métodos de comprobación

Nuestra aplicación siempre seguirá completamente el proceso de comprobación de la equivalencia, que actuará sobre dos expresiones regulares y acabará indicando si éstas son equivalentes o no y mostrando, solo en el segundo caso, un ejemplo de cadena que las diferencie.

Ya con todos los algoritmos que usaremos descritos y entendidos, y conociendo la secuencia de algoritmos que se seguirán en cada caso particular, trataremos de introducir “pereza” en las secuencias que lo permitan. Así, describiremos nuevos algoritmos que siempre empezarán y acabarán en los mismos puntos que la secuencia de algoritmos propuesta inicialmente pero no

necesitando, en aquellos casos en que las expresiones regulares no describan el mismo patrón, de una construcción completa de todos los autómatas.

En esta fase, definiremos claramente los algoritmos, tanto aquellos que seguirán fielmente los estudiados, como aquellos que se mezclarán con otros para producir un algoritmo nuevo y más "perezoso".

1.5.3 – Desarrollo del sistema

En esta fase, crearemos la herramienta de comprobación de la equivalencia de especificaciones basadas en expresiones regulares. Para ello seguiremos varias fases:

1. Implementación de un algoritmo que creará, a partir de una expresión regular escrita en texto simple, la estructura de datos en forma de árbol que se necesita para empezar cualquiera de las secuencias de algoritmos.
2. Implementación de los algoritmos que comprobarán la equivalencia, partiendo de los árboles de las expresiones regulares, y llegando a un resultado legible por el usuario.
3. Implementación de los métodos de entrada de las expresiones regulares a partir de una lista de expresiones y creación de una versión incompleta de la interfaz de usuario.
4. Implementación de un analizador léxico y sintáctico que recoja una especificación basada en expresiones regulares y genere su representación arborescente requerida por nuestro programa. Creación de la versión final de la interfaz de usuario.

1.5.4 – Evaluación preliminar

Para comprobar el correcto funcionamiento de la herramienta, y a lo largo de todas las fases de la implementación, se aplicarán diferentes evaluaciones:

- En las primeras fases, cuando tan solo funcione la creación de los árboles de las expresiones regulares y los algoritmos que comprueben la equivalencia entre estas expresiones, someteremos a los algoritmos a una batería de expresiones regulares que se compararán una a una.
- En las últimas fases, introduciremos especificaciones reales basadas en expresiones regulares y probaremos todas las combinaciones posibles de entre las que dispone la herramienta para la comprobación de su equivalencia. Estas especificaciones serán proporcionadas por nuestro director de TFG a partir de los trabajos prácticos que habitualmente realizan los estudiantes de la asignatura de "Procesadores de Lenguajes".

1.6 – Work plan

The work plan to reach the different objectives is detailed below.

1.6.1 – Study of the algorithms

To approach the first objective, we will investigate the algorithms that allow us to check the equivalence between different regular expressions. For this purpose, the equivalence checking will be reduced to checking equivalence between deterministic finite automaton. In this regard, we have found the Hopcroft-Karp algorithm [11].

These deterministic finite automatons can be obtained from regular expressions by means of several algorithms: Thompson [1], followers [3], derivatives [4], partial derivatives [4] or Berry-Sethi [6]. Some of these algorithms only produce non-deterministic finite automaton, so it will also be necessary to study a subset determination algorithm (in its two versions: with and without λ-transitions) [9].

In this phase, we study all the algorithms that we need to implement the tool.

1.6.2 – Design of checking methods

Our application will always fully follow the equivalence checking process, which will apply to two regular expressions and will end up showing whether they are equivalent or not and displaying, only in the second case, a string example that distinguishes both expressions.

With all the algorithms described and understood, and by knowing the sequence of algorithms that will be applied in each case, we will try to introduce "laziness" in the equivalency-checking sequences. Thus, we will describe new algorithms that will always start and end at the same points as the initially proposed sequences of algorithms, in those cases where the regular expressions do not describe the same pattern, they do not require to complete the construction of all the automaton.

In this phase, we will clearly define the algorithms, both those that will faithfully match the algorithms studied, and those that will be mixed with others to produce a new and "lazier" algorithm.

1.6.3 – System development

In this phase, we will create the tool for checking the equivalence of specifications based on regular expressions. To do so, we will follow several phases:

1. Implementation of an algorithm that will create, from a regular expression written in plain text, the tree-like data structure needed to start any of the sequences of algorithms.
2. Implementation of the algorithms that will check the equivalence, starting from the regular expression trees, and end up with a user-readable result.
3. Implementation of the input methods to obtain the regular expressions from a list of expressions and creation of an incomplete version of the user interface.
4. Implementation of a scanner and a parser that collects a specification based on regular expressions and generates its tree-like representation, required by our program. Creation of the final version of the user interface.

1.6.4 – Preliminary evaluation

To check whether the tool performance is correct, we will apply different evaluations throughout all the phases of the implementation:

- In the first phases, when there are only the creation of regular expression trees and the algorithms that check the equivalence between these data structures, we will test the algorithms against a set of regular expressions that will be compared one by one.
- In the last phases, we will introduce real specifications based on regular expressions and test all the possible combinations available in the tool to check their equivalence.

These specifications will be provided by our degree project supervisor based on the practical work that usually are carried out by the students of the course "Language Processors" at Complutense University of Madrid.

1.7 – Estructura de la memoria

La estructura del resto de la memoria es como sigue:

- En el capítulo 2 se detallan los algoritmos básicos necesarios para realizar la comprobación de equivalencia entre expresiones regulares: la creación de los autómatas finitos, la determinación de autómatas finitos no deterministas (en su versión para AFNs con y sin λ-transiciones) y la comprobación de equivalencia entre dos autómatas a través del algoritmo de Hopcroft-Karp.
- En el capítulo 3 se describen los algoritmos finales que se siguen en la aplicación.
- En el capítulo 4 se habla del diseño y la implementación de la herramienta, dándose detalles de la arquitectura, la interfaz, los archivos de entrada, etc.
- En el capítulo 5 se muestran una serie de especificaciones de prueba y los resultados de comparar cada una de ellas con una especificación concreta. Todas ellas facilitadas por nuestro director de TFG.
- En el capítulo 6 se presentan las conclusiones de este trabajo y posibles vías de continuación.
- En el capítulo 7 se detalla la contribución que ha hecho cada miembro del equipo a este trabajo.

1.8 – Document Structure

In the remainder of the memory, the structure is as follows:

- Chapter 2 details the basic algorithms needed to perform the equivalence checking between regular expressions: the creation of the finite automaton, the determination of non-deterministic finite automaton (in its version for NFAs with and without λ-transitions) and the equivalence checking between two automata through the Hopcroft-Karp algorithm.
- Chapter 3 describes the final algorithms used in the application.
- Chapter 4 discusses the design and implementation of the tool, giving details on its architecture, the interface, the input files, etc.
- Chapter 5 shows a series of test specifications and the results of comparing each of them with a reference specification. All of them provided by our degree project supervisor.
- Chapter 6 presents the conclusions of this work and possible future lines.
- Chapter 7 details the contribution made by each member of the team to this work.

2. Algoritmos para la comprobación de la equivalencia de expresiones regulares

En este capítulo se explicarán todos los algoritmos que hemos estudiado para la realización del proyecto. Comenzaremos con una pequeña introducción con algunas definiciones que es importante conocer para las explicaciones (apartado 2.1). Luego detallaremos, de forma ordenada: los algoritmos para la creación de autómatas (apartado 2.2), la conversión de los autómatas finitos no deterministas en deterministas para aquellos que lo necesiten (apartado 2.3), y, por último, el algoritmo de comprobación de equivalencia (apartado 2.4).

2.1 – Aspectos preliminares

Antes de empezar a describir los diversos algoritmos que se han utilizado para la comprobación de equivalencias, hay algunos términos que hemos considerado conveniente explicar[2]:

- Lenguajes formales: son conjuntos de cadenas finitas formadas mediante un alfabeto y una gramática formalmente especificada.
- Alfabeto: el conjunto no vacío de símbolos que formarán parte de la gramática.
- Gramática: conjunto finito de reglas que, a través de los símbolos del alfabeto, permitirán la formación de cadenas finitas o palabras.
- Palabra: cadena finita de símbolos de un lenguaje formal.
- Autómata finito (AF): modelo matemático de una máquina que acepta o rechaza entradas (cadenas) sobre un alfabeto. Está formado por una serie de estados, transiciones entre ellos (que se harán a través de símbolos del alfabeto o de manera espontánea, en el caso de λ-transiciones), un estado inicial y un conjunto de estados finales o de aceptación.

  Formalmente, desde el punto de vista más general, un AF M queda caracterizado por una tupla <Q, ∑, δ, q0, F> donde:
  - Q es un conjunto finito de estados.
  - ∑ es el alfabeto del lenguaje reconocido por M.
  - δ es una función de transición que, dado un estado q y un símbolo del alfabeto a, devuelve el conjunto posible de estados δ(q,a) al que puede transitarse desde q cuando el siguiente símbolo en la entrada sea a; también se admiten λ-transiciones, y el conjunto de estados al que se transita desde q espontáneamente, sin consumir el siguiente símbolo de la entrada, se denota como δ(q, λ).
  - q0 es el estado inicial, por el que empieza el reconocimiento.
  - F es un conjunto de estados finales o de aceptación (si el autómata llega a alguno de ellos, y no quedan más símbolos en la entrada, se dice que acepta la frase que había en dicha entrada).

[2] Información extraída de https://sg.com.mx/content/view/545 , http://www.fdi.ucm.es/profesor/fpeinado/courses/compiling/repaso-lenguajesformales.pdf y https://webs.ucm.es/info/pslogica/automatas.pdf

- Estado: contenido de la memoria de un autómata finito en un momento concreto del proceso.
- Función de transición: define el proceso en el que, partiendo de un estado y con un símbolo (o no, en el caso de las transiciones vacías) se alcanza otro estado del autómata.
- λ-transición o transición vacía: una transición que no se guía por ningún símbolo y puede seguirse en cualquier momento, siempre y cuando estemos en el estado del que parte.
- Autómata finito no determinista (AFN): es aquel autómata finito que permite, bien la existencia de transiciones vacías, bien varias transiciones con el mismo símbolo que vayan de un mismo estado origen a distintos destinos, o bien ambos tipos de transiciones. Por tanto, el modelo más general de AFN coincide con el modelo más general de AF dado anteriormente, aunque también puede haber modelos de AFNs más específicos (por ejemplo, AFNs que no admiten λ-transiciones).
- Autómata finito determinista (AFD): aquel autómata que no permite ni transiciones vacías ni dos transiciones que, partiendo del mismo estado y siguiendo el mismo símbolo, vayan a dos o más estados destino distintos. Por tanto, la función de transición produce siempre un único estado.
- Expresión regular: cadena de caracteres utilizada en la detección de patrones en textos.
- Metacaracteres: caracteres especiales en los que se apoyan las expresiones regulares. Estos caracteres no se representan a sí mismos (por ejemplo, un asterisco no significa que el patrón incluya, literalmente, un asterisco), sino que tiene su propio significado. Son los que aportan a las expresiones regulares su flexibilidad y potencia.
- Especificaciones basadas en expresiones regulares: descripción de un conjunto de patrones utilizando el formalismo de las expresiones regulares, convenientemente extendido con un conjunto más rico de metacaracteres y con mecanismos básicos de abstracción. Se utilizan, por ejemplo, para especificar el léxico de lenguajes de programación.

A continuación, describimos el formalismo básico en el que se basan las expresiones regulares:

- ∅ es una expresión regular que denota el lenguaje vacío (es decir, el lenguaje que no contiene ninguna cadena).
- λ es una expresión regular que representa un lenguaje que consta únicamente de la cadena vacía (es decir, la cadena que no tiene ningún símbolo del alfabeto, que se representará también por λ).
- Expresión regular *carácter*, representada por cualquier carácter *a* que pueda formar parte de las cadenas del lenguaje. Denota un lenguaje que consta únicamente de la cadena *a*.
- Expresión regular *unión*, representada como α|β o α+β, siendo α y β expresiones regulares. Denota la unión de los lenguajes denotados por α y por β.

- Expresión regular *concatenación*, representada como αβ, siendo α y β expresiones regulares. Denota el lenguaje formado por todas las cadenas de la forma *uv*, donde *u* es una cadena del lenguaje denotado por α, y *v* es una cadena del lenguaje denotado por β.
- Expresión regular *cierre de Kleene*, representada como α*, siendo α una expresión regular. Denota el lenguaje formado por el resultado de concatenar un número arbitrario (incluido el cero) de cadenas del lenguaje denotado por α. Es decir, el lenguaje formado por las cadenas λ, u_0, u_0 u_1, u_0 u_1 u_2 … y así sucesivamente, donde cada u_i es una cadena del lenguaje denotado por α.

2.2 – Transformación de expresiones regulares en autómatas finitos

2.2.1 – Método de Thompson

El nombre del método se debe a su autor K. Thompson, quien lo describió en 1969 [1].

Es un método muy directo e intuitivo que sirve para obtener, a partir de una expresión regular, un autómata finito no determinista con λ-transiciones (λ-AFN) caracterizado por tener un único estado final, a diferencia del generado por cualquiera de los otros métodos. La construcción del autómata se basa en llamadas recursivas a reglas definidas para cada uno de los posibles tipos de expresiones regulares.

A continuación, se detallan las transformaciones de las expresiones regulares a un λ-AFN:

- Para ∅, la expresión regular que describe el lenguaje vacío, el algoritmo produce un único estado inicial sin transiciones, por lo que no se acepta ninguna cadena.
- Para λ, que corresponde al lenguaje { λ }, el autómata que lo reconoce es un estado inicial que transita a un estado final mediante una λ-transición.
- Para cualquier símbolo *a* presente en el alfabeto del lenguaje definido, el reconocimiento se lleva a cabo por un estado inicial que transita a un estado final a través de dicho símbolo.
- Unión de expresiones regulares α+β. Ya con los autómatas de las expresiones α y β construidos (*at* y *bt*, respectivamente), creamos un estado inicial del que parten dos λ-transiciones a los estados iniciales de ambos. Creamos un estado final y luego, a partir de los estados finales de los *at* y *bt* iniciales, una lambda-transición al nuevo estado final. Los estados finales que pertenecían a los *at* y *bt* originales se marcan como no finales.
- Concatenación de expresiones regulares αβ. Ya con los autómatas de las expresiones α y β construidos (*at* y *bt*, respectivamente), marcamos el estado inicial del autómata *at* como el único estado inicial. Luego, creamos una λ-transición que vaya al estado inicial de *bt* desde el estado final de *at* (que marcamos como no final).
- Cierre de Kleene α*. Ya con el autómata de la expresión α construido (*at*), creamos dos estados, uno inicial y uno final. Del nuevo estado inicial sacamos dos λ-transiciones: una hacia el estado inicial de *at* y otra hacia el nuevo estado final. Del estado final de *at*, creamos otras dos λ-transiciones: una hacia el nuevo estado final y otra hacia el estado inicial original de *at*. Convertimos el estado final de *at* en no final.

Obsérvese que el autómata que resulta no está completamente definido en el sentido de que, para cada estado, no se especifican necesariamente transiciones para todos los símbolos. Para obtener un autómata completo basta introducir un estado adicional de error[3] y hacer que el autómata transite a dicho estado para todas las transiciones no especificadas. Asimismo, y como se puede observar en la Figura 1, este método introduce muchas λ-transiciones y, en consecuencia, más estados en el autómata final. Esto complica su posterior procesado en la comprobación de equivalencia. En el algoritmo de seguidores, mostrado en la sección 2.2.2, se explica una versión optimizada del algoritmo que introduce menos λ-transiciones en el autómata.

Este método es, a pesar de su problemática del exceso de λ-transiciones, un algoritmo fácil de implementar comparado con otros que veremos más adelante.

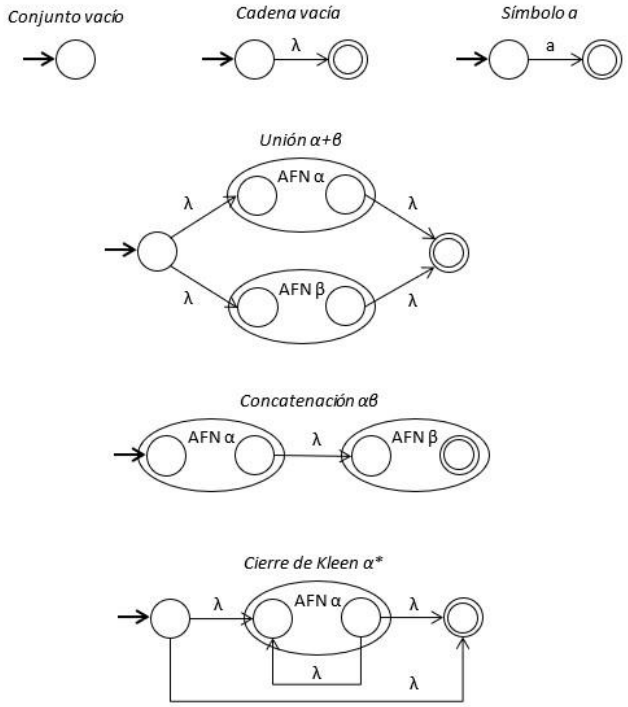

Figura 1: transiciones en algoritmo de Thompson

[3] Es decir, un estado no final desde el que se transite a sí mismo con todos los símbolos del alfabeto de forma que, si el autómata alcanza dicho estado, es seguro que rechazará la entrada.

Como ejemplo, aplicaremos las reglas a la expresión regular ba+a*, obteniendo así el autómata de la Figura 2.

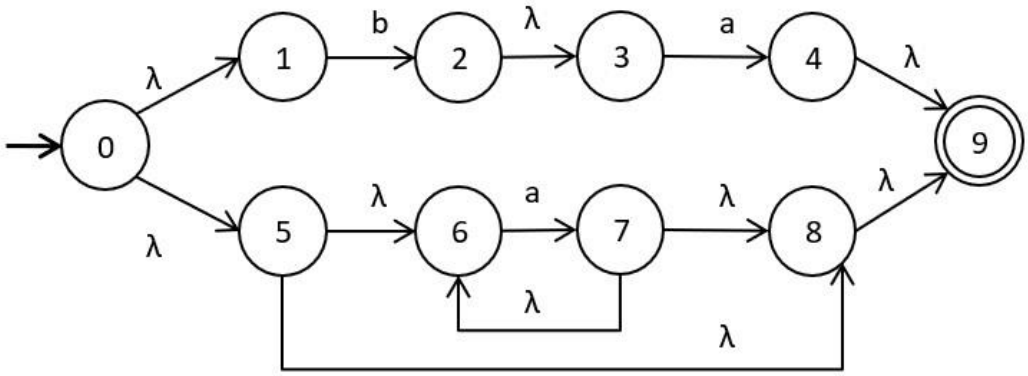

Figura 2: autómata resultante de aplicar el método de Thompson a ba+a* *(se omiten las transiciones al estado de error)*

2.2.2 – Método de los seguidores

La construcción mediante el método de Thompson puede mejorarse en el número de estados y transiciones. Este algoritmo es una de las soluciones que se plantearon al respecto. Ya en 1988, S. Sippu y E. Soisalon-Soininen [2] propusieron una construcción que reduce las λ-transiciones en todos los casos recursivos de las reglas. Sin embargo, no las elimina del todo. Las optimizaciones propuestas por L. Ilie y S. Yu en 2003 [3] dentro de este algoritmo lo consigue.

Las mejoras son las siguientes, representadas gráficamente en la Figura 3:

- ∅, el algoritmo produce un único estado inicial sin transiciones.
- λ, se produce un único estado final e inicial a la vez.
- Para *a*, se produce un estado inicial que va a uno final a través del símbolo *a*.
- Unión de expresiones regulares α+β. Se fusionan el estado inicial del autómata de α con el del autómata de β, manteniendo las transiciones de ambos en el nuevo estado. En caso de que alguno sea también estado final, el estado resultante también lo será.
- Concatenación de expresiones regulares αβ. Se combinan los estados finales del primer autómata con el estado inicial del segundo. El estado resultante es final solo si el estado inicial del autómata de β lo es.
- Cierre de Kleene α*. Se añaden las transiciones que salen del estado inicial a todos los estados finales del autómata de α y, el estado inicial se vuelve final.

Al igual que en el algoritmo de Thompson, las transiciones omitidas en el autómata resultante pueden completarse transitando a un estado de error.

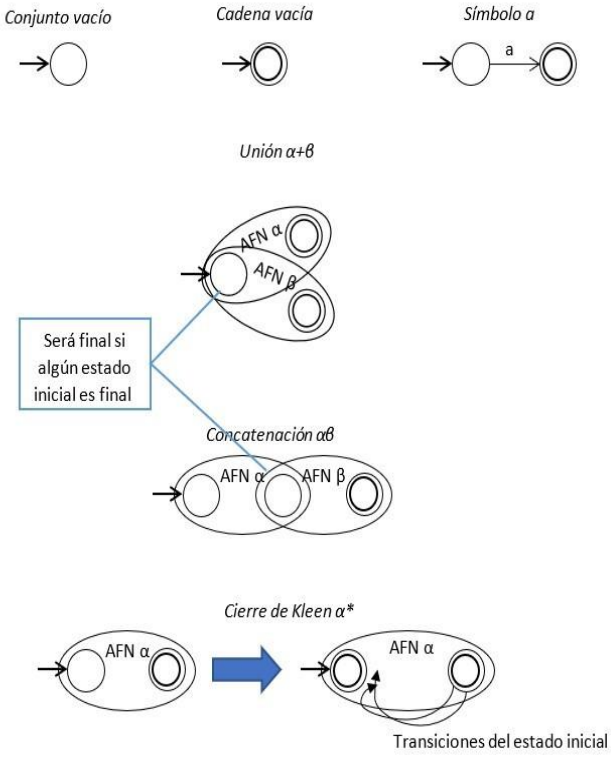

Figura 3: transiciones del método de los seguidores

Usando como ejemplo la expresión regular ba+a*, el AFN resultante de este método sería el de la Figura 4.

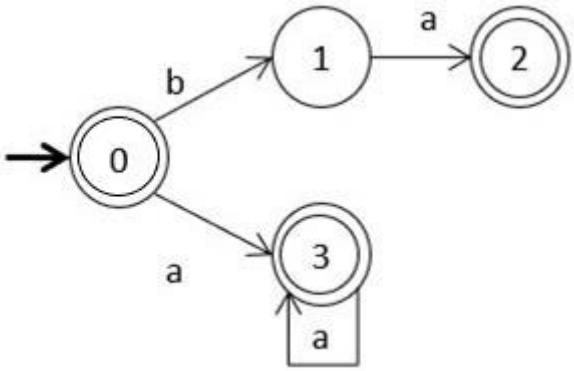

Figura 4: autómata construido mediante el método de los seguidores a partir de ba+a* *(se omiten transiciones al estado de error)*

En el peor caso, las transiciones que se añaden en las transformaciones pueden escalarse a un orden cuadrático con relación al tamaño de la expresión regular, pero este hecho queda compensado al obtenerse un AFN sin λ-transiciones, dado que ya no se hace necesario un paso previo de eliminación de λ-transiciones como el de lambda-cierre.

2.2.3 – Método de las derivadas

Propuesto por Brzozowski en 1.964 [4], se utiliza para crear autómatas finitos deterministas a partir de expresiones regulares.

Una derivada de una expresión α con respecto al símbolo *s* es otra expresión regular que denota el lenguaje que resulta de tomar todas aquellas cadenas del lenguaje para α que comienzan por *s,* y eliminar *s* del comienzo de dichas cadenas. Por tanto, si consideramos un estado para iniciar el reconocimiento de α, desde este estado podremos transitar, por *s*, a otro estado asociado al reconocimiento de la derivada de α con respecto a *s*. Formalmente, esta derivada se denota por: $\delta_s(\alpha)$.

Para calcular la derivada, se sigue un procedimiento de sustitución por derivadas con casos base y casos recursivos[4]:

Casos base:

- $\delta_s(\emptyset) = \emptyset$
- $\delta_s(\lambda) = \emptyset$
- $\delta_s(s) = \lambda$
- $\delta_s(b) = \emptyset$ si $b \neq s$

Casos recursivos:

- $\delta_s(\alpha+\beta) = \delta_s(\alpha) + \delta_s(\beta)$
- $\delta_s(\alpha\beta) = \delta_s(\alpha)\beta + \varepsilon$
    - Si α puede producir la cadena vacía λ: $\varepsilon = \delta_s(\beta)$
    - En otro caso: $\varepsilon = \emptyset$
- $\delta_s(\alpha^*) = \delta_s(\alpha)\alpha^*$

Utilizando derivadas podemos generar un autómata equivalente a una expresión regular α de la siguiente manera:

- Cada estado del autómata tiene asociado una expresión regular.
- El estado inicial tiene asociada la expresión regular α.
- Si hay un estado que tiene asociada una expresión regular β, para cada símbolo del alfabeto *s* y para cada derivada $\delta_s(\beta)$, habrá un único estado que tiene asociada una expresión regular γ *equivalente* a $\delta_s(\beta)$. Además, habrá una transición etiquetada por *s* que va desde el estado para β al estado para γ.
- Los estados finales del autómata serán aquellos cuyas expresiones regulares asociadas denoten lenguajes que contengan la cadena vacía λ.

Es fácil comprobar que el autómata así construido es un autómata finito determinista mínimo (es decir, con el mínimo número de estados posible) equivalente a α.

[4] Estas reglas recuerdan a las reglas de derivación en Cálculo, motivo por el que las expresiones resultantes se denominan *derivadas* de las expresiones originales.

A modo de ejemplo, describamos cómo aplicar el método para construir un AFD equivalente a la expresión ba+a*, con el alfabeto = {a,b}):

1. Creamos un estado inicial con la expresión regular inicial (Figura 5).

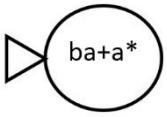

Figura 5: estado inicial (Derivadas)

2. Calculamos las derivadas δ_a(ba+a*) y δ_b(ba+a*):
   - δ_a (ba + a*) = δ_a (ba) + δ_a (a*) = Ø + a* = a*
     - δ_a (ba) = δ_a (b)a + Ø = Øa + Ø = Ø
       - δ_a (b) = Ø
     - δ_a (a*) = δ_a (a)a* = λa* = a*
       - δ_a (a) = λ
   - δ_b (ba + a*) = δ_b (ba) + δ_b (a*) = a + Ø = a
     - δ_b (ba) = δ_b (b)a + Ø = λa + Ø = a
       - δ_b (b) = λ
     - δ_b (a*) = δ_b (a)a* = Øa* = Ø
       - δ_b (a) = Ø
3. Dado que no hay ningún estado cuya expresión sea equivalente a *a** ni a *a*, creamos dos nuevos estados asociados a dichas expresiones, así como las correspondientes transiciones desde el estado inicial (Figura 6).

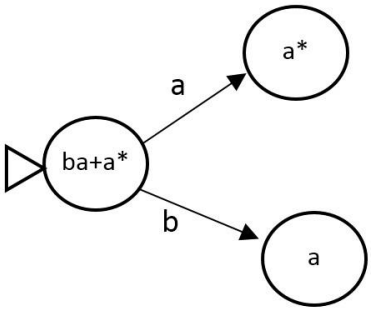

Figura 6: primeras transiciones

4. Repetimos el proceso con todas y cada una de las expresiones regulares que vayan surgiendo y no hayamos explorado.
   - δ_a (a) = λ
   - δ_b (a) = Ø
   - δ_a (a*) = δ_a (a)a* = λa* = a*
   - δ_b (a*) = δ_b (a)a* = Øa* = Ø
   - δ_a (λ)= δ_b (λ)= δ_a (Ø)= δ_a (Ø) = Ø

5. Marcamos como estados finales aquellos marcados con una expresión regular que denote un lenguaje que contenga a λ. En este ejemplo, tanto a* como λ son estados finales, también lo es el estado inicial.

La Figura 7 muestra el AFD resultante.

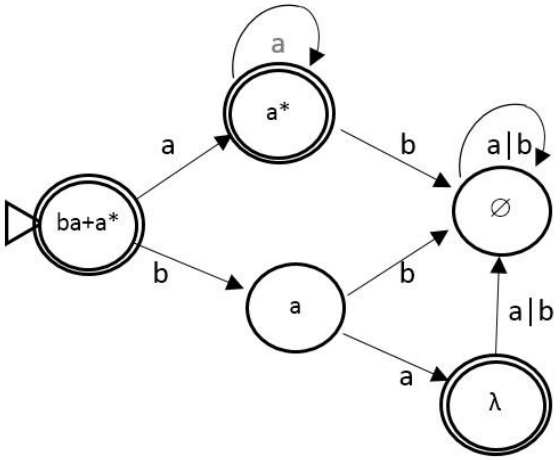

Figura 7: resultado algoritmo derivadas para ba+a*

El principal problema del método de las derivadas es que, además de que a veces, y dependiendo de la expresión regular, las derivadas puedes ser muy complejas, en su propia formulación es necesario comprobar la equivalencia de expresiones regulares, que es precisamente el problema que queremos resolver. Para evitar esta dificultad puede relajarse el criterio de equivalencia por otro fácilmente comprobable. A este respecto:

- Una posibilidad puede ser utilizar la *identidad* entre expresiones regulares como criterio (es decir, cuando se genera una derivada que no se ha generado anteriormente, se genera un nuevo estado). El problema aquí es que, en expresiones en las que aparece *, el número de derivadas que se generará es infinito. Por tanto, esta posibilidad no es factible.
- Otra posibilidad es sustituir el criterio de *equivalencia* por uno más simple, denominado *congruencia módulo* ACI-*similitud* [4]. Para ello, las expresiones regulares se *normalizan* aplicando las propiedades asociativa, conmutativa e idempotente (es decir, $\alpha+\alpha=\alpha$) de la unión, de tal forma que dos expresiones son ACI *congruentes* cuando normalizan a la misma expresión. En este caso, el número de posibles derivadas ACI normalizadas es finito, por lo que el método funciona, al menos en teoría. Asimismo, pueden aplicarse otras reglas de simplificación de expresiones regulares para reducir el número de estados.

2.2.4 – Método de las derivadas parciales

Propuesto por Antimirov en 1.996 [4] y optimizado en 2.002 por Champarnaud & Ziadi [5], permite la construcción de autómatas finitos no deterministas con un bajo número de estados basándose en el algoritmo de las derivadas propuesto por Brzozowski.

Las derivadas parciales de una expresión regular α con respecto a un símbolo *s* son un conjunto de expresiones $\beta_0 \dots \beta_n$ que (i) no son de la forma $\gamma+\phi$; y (ii) $\beta_0 + \dots + \beta_n$ es equivalente a $\delta_s(\alpha)$. De esta forma, las derivadas parciales pueden entenderse como los términos de una normalización de las derivadas, expresadas como sumas de términos más simples. En este caso, si tenemos un estado que inicia el reconocimiento de α, cuando veamos *s* podremos seguir el

reconocimiento a partir de cualesquiera de los estados asociados con β_i. Como consecuencia, en este caso el autómata resultante será, normalmente, no determinista. La ventaja, sin embargo, es que el número de derivadas parciales posibles es muy reducido.

Formalmente, mediante $\partial_s(\alpha)$ denotaremos el conjunto de derivadas parciales de α con respecto a *s*. Igual que en el algoritmo de las derivadas, el cálculo de $\partial_s(\alpha)$ se basa en una serie de casos base y casos recursivos[5]:

Casos base:

- ∂_s (Ø) = {Ø}
- ∂_s (λ) = {Ø}
- ∂_s (*s*) = {λ}
- ∂_s (b) = {Ø} si b ≠ s

Casos recursivos:

- ∂_s (α+β) = ∂_s (α) U ∂_s (β)
- ∂_s (αβ) = ∂_s (α)β U ε
    - Si α puede producir la cadena vacía λ: ε = ∂_s (β)
    - En otro caso: ε = Ø
- ∂_s (α*) = ∂_s (α)α*

De esta forma, utilizando derivadas parciales podemos construir un autómata equivalente utilizando un algoritmo muy parecido al de las derivadas. No obstante, en este caso y como ya hemos comentado, el autómata resultante será, normalmente, no determinista. El algoritmo en sí es como sigue:

- Como en el algoritmo de las derivadas, cada estado del autómata tiene asociado una expresión regular.
- También como ocurre con el algoritmo de las derivadas, el estado inicial tiene asociado la expresión regular α.
- Si hay un estado que tiene asociada una expresión regular β, para cada símbolo del alfabeto *s* y para cada derivada parcial γ en $\partial_s(\beta)$, habrá un estado para γ y una transición etiquetada por *s* con origen el estado para β y destino el estado para γ. Es aquí, y porque $\partial_s(\beta)$ puede tener más de una expresión, donde surge el no determinismo del autómata.
- Por último, e igual que en el algoritmo de las derivadas, los estados finales del autómata serán aquellos cuyas expresiones regulares asociadas denoten lenguajes que contengan la cadena vacía λ.

[5] En estas reglas se utiliza la siguiente notación: si Γ es un conjunto de expresiones, y α es una expresión, $\Gamma\alpha$ es el conjunto que resulta de concatenar α a cada expresión en Γ.

Como ejemplo, pondremos la misma expresión que usamos para explicar el algoritmo de Brzozowski: ba + a*:

1. Se crea un estado inicial marcado con la expresión regular original (Figura 8).

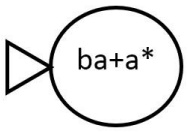

Figura 8: estado inicial (Derivadas parciales)

2. Calculamos las derivadas parciales de esta expresión con respecto de *a* y con respecto de *b*:
   - ∂_a (ba + a*) = ∂_a (ba) U ∂_a (a*) = {Ø} U {a*} = {Ø, a*}
     - ∂_a (ba) = ∂_a (b)a U Ø = {Ø}a U Ø = {Ø}
       - ∂_a (b) = {Ø}
     - ∂_a (a*) = ∂_a (a)a* = {λ}a* = {a*}
       - ∂_a (a) = {λ}
   - ∂_b (ba + a*) = ∂_b (ba) U ∂_b (a*) = {a} U {Ø} = {a, Ø}
     - ∂_b (ba) = ∂_b (b)a U Ø = {λ}a U Ø = {a}
       - ∂_b (b) = {λ}
     - ∂_b (a*) = ∂_b (a)a* = {Ø}a* = {Ø}
       - δ_b (a) = {Ø}

3. Creamos un estado para cada una de las derivadas parciales obtenidas (∅, *a*, *a**) y añadimos las correspondientes transiciones (Figura 9).
   Aquí podemos apreciar la mayor diferencia con el algoritmo de las derivadas de Brzozowski: el no determinismo. En este ejemplo, desde el estado para ba + a* se transita, por *a*, tanto al estado para *a** como al estado para ∅, y por *b* tanto al estado para *a* como al estado para ∅.

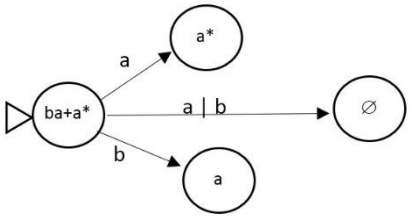

Figura 9: primeras transiciones (Derivadas parciales)

4. Repetimos los pasos 2, 3 y 4 con cada uno de los nuevos nodos hasta que no aparezca ninguno nuevo.
   - ∂_a(a) = {λ}
   - ∂_b(a) = {Ø}
   - ∂_a(a*) = ∂_a(a)a* = {λ}a* = {a*}
   - ∂_b(a*) = ∂_b(b)a* = {Ø}a* = {Ø}
   - ∂_a(λ)= ∂_b(λ)= ∂_a(Ø)= ∂_b(Ø) = {Ø}

5. Marcamos como estados finales aquellos en los que el lenguaje denotado por la expresión regular incluya a λ. El autómata resultante se muestra en la Figura 10.

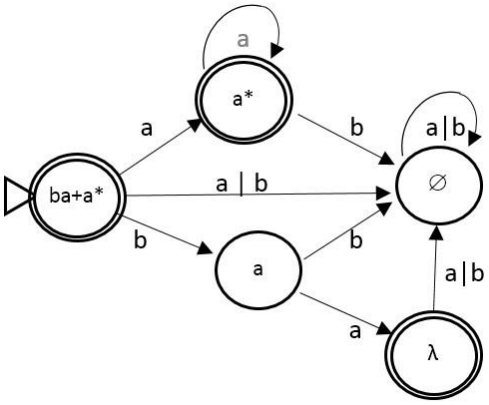

Figura 10: resultado algoritmo derivadas parciales para ba+a*

2.2.5 – Método de Berry-Sethi

Descrito en 1986 por G. Berry y R. Sethi [6], y basándose en el método de las derivadas [4], este método crea un autómata finito no determinista sin λ-transiciones a partir de una expresión regular.

El método de Berry-Sethi comienza *marcando* cada ocurrencia de cada símbolo *s* en la expresión regular mediante un subíndice distinto. Por ejemplo, la expresión ba + a* se marca como $b_0a_0 + a_1$*. Cada símbolo marcado (b_0, a_0 y a_1 en este caso) representa una *posición* en la expresión original. Es posible aplicar, entonces, el algoritmo de las derivadas a la expresión marcada. En este caso, habrá exactamente una derivada distinta de ∅ para cada posición, lo que permite identificar las posiciones (y, por tanto, los estados en el autómata) con dichas derivadas. Asimismo, si en dicho autómata se sustituye, en las etiquetas de las transiciones, cada símbolo a_i por *a*, el autómata resultante (que, en general, ya no será determinista) aceptará el lenguaje denotado por la expresión regular original (véase Figura 11 para ver los posibles autómatas resultantes).

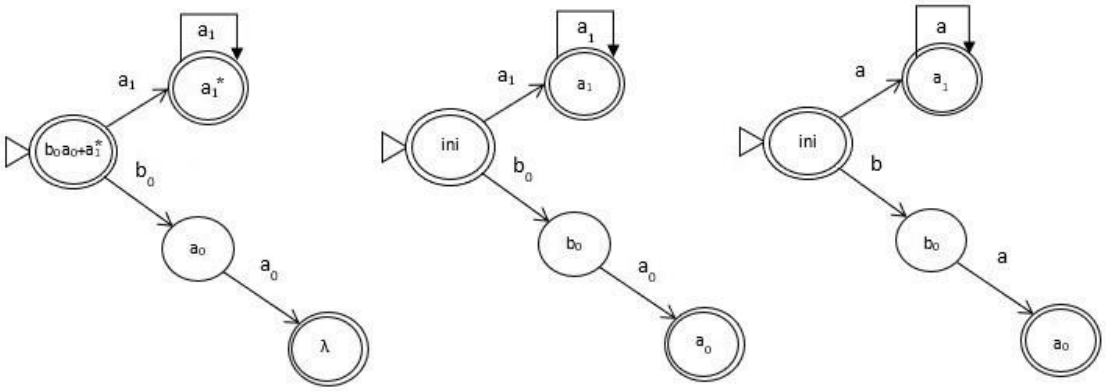

Figura 11: (Izquierda) aplicación del método de las derivadas a la expresión marcada $b_0a_0 + a_1$. (Centro): sustitución de derivadas por las posiciones que las originan. (Derecha): autómata de posiciones que resulta de sustituir cada a_i por a. En todos los casos se omiten transiciones al estado de error.*

Dado que, a excepción del estado inicial y del estado de error $\varnothing$, los estados del autómata construido por el método de Berry-Sethi se corresponden con posiciones, dicho autómata se denomina *autómata de posiciones*. Asimismo, es posible simplificar su cálculo mediante la introducción de los siguientes conceptos:

- El conjunto de *primeros* de una expresión regular α, que son los símbolos por los que comienzan las cadenas del lenguaje para α.
- El conjunto de *sucesores* de un símbolo *a* en α, que son los símbolos que pueden seguir a *a* en las cadenas del lenguaje para α.

A continuación, se analizan dos estrategias de implementación del método, una básica, y una mejorada.

2.2.5.1 – Algoritmo básico

Una forma directa para construir el autómata de *posiciones* es mediante un procesamiento recursivo, en post-orden y ascendente del árbol de sintaxis abstracta de la expresión regular marcada [7]. Para ello, con cada expresión regular marcada α se asocian los siguientes atributos:

- Conjunto de primeros de α (*first*).
- Conjunto de posiciones de α asociadas con estados de aceptación (*last*).
- Relación *sucesores* (*follows*): un conjunto de pares de posiciones. (a_i,b_j) estará en esta relación si y solo si b_j es un sucesor de a_i en α.
- Nulo (*null* $\in$ {true, false}): indica si λ está incluida en el lenguaje denotado por α.

A continuación, se detallan reglas recursivas para asociar con una expresión marcada la tupla de atributos [(*first*, *last, follows*, *null*)]. En estas reglas, $\alpha \rightarrow \Gamma$ significa que la expresión regular α tiene asociada la tupla producida por la combinación funcional Γ. Las reglas en sí son como sigue:

- $\varnothing \rightarrow [(\varnothing, \varnothing, \varnothing,$ **false**$)]$
- $\lambda \rightarrow [(\varnothing, \varnothing, \varnothing,$ **true**$)]$
- $a \rightarrow [(\{a\}, \{a\}, \varnothing,$ **false**$)]$
- $\alpha\beta \rightarrow$ **let**
  $\alpha \rightarrow [(\text{first}_\alpha, \text{last}_\alpha, \text{follows}_\alpha, \text{null}_\alpha)]$
  $\beta \rightarrow [(\text{first}_\beta, \text{last}_\beta, \text{follows}_\beta, \text{null}_\beta)]$
  $\text{first}_{\alpha\beta} = \text{first}_\alpha \cup$ (**if** null_α **then** first_β **else** $\varnothing$ **fi**)
  $\text{last}_{\alpha\beta} = \text{last}_\beta \cup$ (**if** null_β **then** last_α **else** $\varnothing$ **fi**)
  $\text{follows}_{\alpha\beta} = \text{follows}_\alpha \cup \text{follows}_\beta \cup (\text{last}_\alpha \times \text{first}_\beta)$
  in
  $[(\text{first}_{\alpha\beta}, \text{last}_{\alpha\beta}, \text{follows}_{\alpha\beta}, \text{null}_\alpha \wedge \text{null}_\beta)]$
- $\alpha+\beta \rightarrow$ **let**
  $\alpha \rightarrow [(\text{first}_\alpha, \text{last}_\alpha, \text{follows}_\alpha, \text{null}_\alpha)]$
  $\beta \rightarrow [(\text{first}_\beta, \text{last}_\beta, \text{follows}_\beta, \text{null}_\beta)]$
  $\text{first}_{\alpha+\beta} = \text{first}_\alpha \cup \text{first}_\beta$
  $\text{last}_{\alpha+\beta} = \text{last}_\alpha \cup \text{last}_\beta$
  $\text{follows}_{\alpha+\beta} = \text{follows}_\alpha \cup \text{follows}_\beta$
  in
  $[(\text{first}_{\alpha+\beta}, \text{last}_{\alpha+\beta}, \text{follows}_{\alpha+\beta}, \text{null}_\alpha \vee \text{null}_\beta)]$

- $\alpha^* \rightarrow$ **let** $\alpha \rightarrow [(first_\alpha, last_\alpha, follows_\alpha, null_\alpha)]$
  $follows_{\alpha^*} = follows_\alpha \cup (last_\alpha \times first_\alpha)$

  in

  $[(first_\alpha, last_\alpha, follows_{\alpha^*}, \textbf{true})]$

A modo de ejemplo, sea la expresión regular (ab + λ)b*. La correspondiente expresión marcada es $(a_0b_0 + \lambda)b_1^*$. Aplicando las reglas anteriores a dicha expresión marcada, se tiene que:

- $(a_0b_0 + \lambda)b_1^* \rightarrow [(\{a_0,b_1\}, \{b_0,b_1\},\{ (a_0, b_0), (b_1, b_1),(b_0, b_1)\} ,\textbf{true})]$
  - $(a_0b_0 + \lambda) \rightarrow [(\{a_0\},\{b_0\} ,\{ (a_0, b_0)\} ,\textbf{true})]$
    - $a_0b_0 \rightarrow [(\{a_0\},\{b_0\} ,\{ (a_0, b_0)\} ,\textbf{false})]$
      - $a_0 \rightarrow [(\{a_0\},\{a_0\} ,\varnothing ,\textbf{false})]$
      - $b_0 \rightarrow [(\{b_0\},\{b_0\} ,\varnothing ,\textbf{false})]$
    - $\lambda \rightarrow [(\varnothing,\varnothing , \varnothing ,\textbf{true})]$
  - $b_1^* \rightarrow [(\{b_1\},\{b_1\} ,\{(b_1, b_1)\} ,\textbf{false})]$
    - $b_1 \rightarrow [(\{b_1\},\{b_1\} ,\varnothing ,\textbf{false})]$

Una vez procesada la expresión regular de forma recursiva, se obtiene toda la información necesaria para construir el autómata. Para ello:

- Se crea un estado inicial, uno por cada posición en la expresión marcada, y un último asociado a ∅. Los estados asociados a las posiciones en *last* se marcan como de aceptación. Asimismo, si la expresión marcada es anulable, se marca también el estado inicial como de aceptación.
- Se crea una transición desde el estado inicial a cada estado asociado con cada posición a_i en el conjunto *first*, etiquetada por *a* (el símbolo sin marcar).
- Para cada par (a_i, b_j) en *follows*, se crea una transición con origen el estado para a_i y destino el estado para b_j. Dicha transición se etiqueta con *b*.
- Para cada estado y cada símbolo *a*, si de dicho estado no parte ninguna transición etiquetada con *a*, se crea una transición desde dicho estado al asociado a ∅ y se etiqueta con *a*.

Este método realiza, por tanto, un pre-proceso de la expresión completando los atributos pertinentes para luego construir el autómata a partir de la información recabada. Su principal inconveniente radica en que, por ejemplo, la aplicación de la regla para expresiones de la forma αβ involucra un factor cuadrático en el cálculo de $follows_{\alpha\beta}$ (debido al término $last_\alpha \times first_\beta$, que implica construir todos los pares de la forma (a_i, b_j), con a_i una posición en $last_\alpha$ y b_j una posición en $first_\beta$). Lo mismo ocurre con la aplicación de la regla a expresiones de la forma *a**. Por tanto, el algoritmo resultante tendrá un coste $O(n^3)$ (con *n* el número de nodos en el árbol de sintaxis abstracta de la expresión).

2.2.5.2 – Algoritmo mejorado

Es posible conseguir un algoritmo $O(n^2)$ realizando dos recorridos del árbol de sintaxis abstracta de la expresión regular marcada. Este algoritmo fue el originalmente descrito por Berry y Sethi [6], y se encuentra también re-elaborado, por ejemplo, en [8]. En este algoritmo:

- En un primer recorrido se computan los atributos *first, last* y *null* para cada nodo, siguiendo una estrategia análoga a la descrita en la sección anterior. Si α es una expresión regular marcada, mediante $first_\alpha$, $last_\alpha$ y $null_\alpha$ denotaremos, respectivamente, los atributos *first*, *last* y *null* asociados con α.
- En un segundo recorrido se propaga un atributo *heredado* (una *entrada*), también llamado *follows*, que contiene los símbolos que pueden aparecer tras las cadenas denotadas por la expresión regular. Por tanto, cada vez que se llega a un nodo hoja asociado con una posición, pueden crearse directamente las transiciones que parten del estado asociado al mismo. Este hecho evita el factor cuadrático de cálculos como $last_\alpha \times first_\beta$ y $last_\alpha \times first_\alpha$, y que aumentaban en un orden la complejidad del algoritmo básico descrito en la sección anterior.

A continuación, se describe cómo determinar el atributo heredado *follows* para cada componente de una expresión marcada en función del atributo heredado de dicha expresión. Para ello, se utiliza la notación $\mathbf{F}\ \alpha \rightarrow \Gamma$, donde: (i) **F** es el valor de *follows* para α; y (ii) Γ es una combinación funcional que produce un conjunto de elementos $\mathbf{F_i}\ \alpha_i$ indicando el conjunto *follows* $\mathbf{F_i}$ para cada componente α_i de α:

- $\mathbf{F}_{\alpha\beta}\ \alpha\beta \rightarrow$ **let** $\mathbf{F}_\alpha = first_\beta \cup$ (**if** $null_\beta$ **then** $\mathbf{F}_{\alpha\beta}$ **else** $\varnothing$)
  in
  { $\mathbf{F}_\alpha\ \alpha$, $\mathbf{F}_{\alpha\beta}\ \beta$ }
- $\mathbf{F}_{\alpha+\beta}\ \alpha+\beta \rightarrow$ { $\mathbf{F}_{\alpha+\beta}\ \alpha$, $\mathbf{F}_{\alpha+\beta}\ \beta$ }
- $\mathbf{F}_{\alpha^*}\ \alpha^* \rightarrow$ { $\mathbf{F}_{\alpha^*} \cup first_\alpha\ \alpha$}

A modo de ejemplo, se muestran los conjuntos *follows* para cada sub-expresión de la expresión marcada $(a_0b_0 + \lambda)b_1^*$:

- **{}** $(a_0b_0 + \lambda)b_1^*$
  - **{b_1}** $(a_0b_0 + \lambda)$
    - **{b_1}** a_0b_0
      - **{b_0}** a_0
      - **{b_1}** b_0
    - **{b_1}** λ
  - {} b_1^*
    - **{b_1}** b_1

De esta forma, el coste realizado en cada nodo en ambos recorridos del algoritmo es lineal, por lo que la complejidad resultante será $O(n^2)$. Este es el motivo por el que en este TFG nos hemos decantado por este algoritmo mejorado para implementar el método de Berry-Sethi.

2.3 – Transformación de AFNs en AFDs: método de los subconjuntos

Como ya dijimos en el apartado 1.5.1, el algoritmo de Hopcroft-Karp que se describirá en el apartado 2.4 necesita, como entrada, autómatas finitos deterministas. Por esto y porque alguno de los algoritmos descritos en el apartado 2.2 devuelven autómatas finitos no deterministas, nos hemos visto obligados a estudiar algoritmos que realicen la transformación de AFNs a AFDs.

2.3.1 – AFNs sin lambda-transiciones

El algoritmo estándar y más conocido es el método de la "construcción por subconjuntos", descrito, por ejemplo, en [9]. La idea es identificar cada subconjunto de estados en el que puede estar el AFN simultáneamente en cada fase de reconocimiento con un estado del AFD equivalente.

Generalmente, el AFD resultante es de mayor tamaño que el AFN original, tanto en número de estados como de transiciones.

El número de subconjuntos posibles está en orden de 2^n, siendo n el tamaño del AFN original. Para paliar esta complejidad exponencial solo se añaden aquellos subconjuntos que realmente son accesibles.

Primero, detallaremos un algoritmo que funciona sobre AFNs que no tienen λ-transiciones. De esta forma, dado un autómata finito no determinista (AFN) $M = < Q, \sum, \delta, q_0, F >$ sin λ-transiciones, es posible obtener un autómata finito determinista (AFD) $M' = <Q', \sum, \delta', \{q_0\}, F'>$ que reconozca exactamente el mismo lenguaje que M como se muestra en el Algoritmo 1:

```
El alfabeto(∑) de M' es el mismo que el de M. El estado inicial es el
subconjunto {q0}
Inicializar el conjunto de Q' con el subconjunto {q0}.
Creamos una pila inicializada con el subconjunto {q0}.
Mientras la pila no esté vacía:
   Desapilar un elemento R de la pila.
   Para cada símbolo a  del alfabeto(∑):
      Encontrar el conjunto de estados R' = Uq∈R δ(q, a).
      Si R' no está en Q'
          añadir R' a Q'
          apilar R'
          Si R' contiene algún estado final, entonces añadir R' a F'
      Fin Si
      Añadir una entrada δ'(R, a)= R' en δ'
```

Algoritmo 1: determinación de AFNs sin lambda-transiciones

Se trata de un algoritmo simple en el que la complejidad, en el peor caso y como ya se ha comentado, es exponencial en el número de estados del AFN (ver, por ejemplo, [10]). Para aliviar esta complejidad, puede introducirse "pereza" al algoritmo, tal y como puede verse con más detalles en la sección 3.4. La idea que permite la pereza es convertir este algoritmo en una subrutina que solo expanda un estado cuando se precise (así, no tiene que hacerse necesariamente la determinación completa).

2.3.2 – AFNs con lambda-transiciones

Cuando el AFN tiene λ-transiciones es necesario añadir a cada subconjunto todos aquellos estados a los que puede transitarse mediante estas λ-transiciones. Para ello se introduce una operación *LAMBDA-CIERRE* sobre conjuntos de estados del AFN.

De esta forma, *LAMBDA-CIERRE(R)* incluirá a *R*, y también al conjunto de estados que se pueden alcanzar mediante λ-transiciones desde los estados de *R*. Un posible pseudocódigo para la implementación de una función que calcule el *LAMBDA-CIERRE* puede ser el mostrado en el Algoritmo 2:

```
Dado un conjunto de estados R del AFN M = < Q, ∑, δ, q0, F >, para obtener
LAMBDA-CIERRE(R):
   LAMBDA-CIERRE(R) = R
   Se apilan todos los estados de R en una pila.
   Mientras la pila no esté vacía:
     Se desapila un estado q de la pila.
     Para todo q' que esté en δ(q, λ):
        Si q' ∉ LAMBDA-CIERRE(R)
           Se añade q' al LAMBDA-CIERRE(R)
           Se apila q'
```

Algoritmo 2: operación de lambda-cierre

Con ello, el algoritmo de los subconjuntos puede extenderse para tratar con AFNs con λ-transiciones como muestra el Algoritmo 3 (recalcar que este algoritmo, dado un AFN M = < Q, ∑, δ, q_0, F>, que esta vez sí puede contener λ-transiciones, obtiene un autómata finito determinista (AFD) M' = < Q', ∑, δ', q'_0, F' > equivalente).

```
El alfabeto(∑)de M' es el mismo que el de M. Su estado inicial q'0 es
LAMBDA-CIERRE({q0}).
Inicializar el conjunto de Q' con el subconjunto LAMBDA-CIERRE({q0}).
Creamos una pila inicializada con el subconjunto LAMBDA-CIERRE({q0}).
Mientras la pila no esté vacía:
   Desapilar un elemento R de la pila.
   Para cada símbolo a del alfabeto(∑):
      Encontrar el conjunto de estados R' = LAMBDA-CIERRE(Uq∈R δ(q, a)).
      Si R' no está en Q'
          añadir R' a Q'
          apilar R'
          Si R' contiene algún estado final, entonces añadir R' a F'
      Fin Si
      Añadir una entrada δ'(R, a)= R' en δ'
```

Algoritmo 3: determinación de AFNs con lambda-transiciones

El algoritmo es, por tanto, idéntico al anterior, con la salvedad de que se aplica la operación de *LAMBDA-CIERRE* a cada subconjunto obtenido.

2.4 – Algoritmo de Hopcroft-Karp para la Comprobación de la Equivalencia de dos AFD

Dos autómatas son equivalentes entre sí si ambos reconocen exactamente las mismas sentencias.

Los métodos que existen para comprobar si dos autómatas son equivalentes o no se basan, en mayor o menor medida, en el algoritmo descrito por John E. Hopcroft y R. M. Karp [11]. Este algoritmo solo puede aplicarse (salvo modificaciones) sobre autómatas finitos deterministas.

Este método, cuyo pseudocódigo se encuentra detallado en Algoritmo 4, consiste en tres reglas que se deben cumplir para que dos autómatas sean equivalentes:

- Los estados iniciales de ambos autómatas deben ser equivalentes.
- Para que dos estados sean equivalentes, y por cada símbolo del alfabeto, debe existir una transición que conduzca a otro par de estados equivalentes entre sí.
- Ningún estado final puede ser equivalente a un estado no final.

Aparte de comprobar la equivalencia entre estados, el algoritmo asocia con los estados *q* una *clase de equivalencia*: conjunto de estados que se han supuesto ya equivalentes a *q*. Esto elimina redundancias en los cálculos.

Dados dos AFD $M_0 = < Q, \sum, \delta, q_0, F >$ y $M_1 = < Q', \sum, \delta', q'_0, F' >$:

```
Para cada estado q de M0 y M1 se fija la clase de equivalencia de q como {q}
Sean q0 y q'0 los estados iniciales de, respectivamente, M0 y M1:
   Si uno de ellos es final y el otro no, devolver "no equivalentes"
   Registrar (q0,q'0) como objetivo a explorar
   Fijar la clase de equivalencia de q0 y  q'0 a {q0,  q'0}
   Mientras haya objetivos a explorar:
        Extraer uno de estos objetivos, (q,q') del registro
        Para cada símbolo a:
                  Si δ(q,a) es final y δ'(q',a) es no final, o a la inversa,
                  devolver "no equivalentes".
                  En otro caso
                       Sean Cδ(q,a) y Cδ'(q',a) las clases de equivalencia de,
                       respectivamente, δ(q,a) y δ'(q',a)
                       Si Cδ(q,a) ≠ Cδ'(q',a)
                         Registrar (δ(q,a), δ'(q',a)) como objetivo a explorar
                         Para cada estado q'' en Cδ(q,a) ∪ Cδ'(q',a)
                            Fijar Cδ(q,a) ∪ Cδ'(q',a) como la clase de
                            equivalencia de q''
    Devolver "equivalentes"
```

Algoritmo 4: Hopcroft-Karp

3. Diseño de comprobadores de equivalencia de expresiones regulares

Como ya hemos mencionado, una vez estudiados los diferentes algoritmos para la creación de autómatas, su determinación y la comprobación de la equivalencia entre dos autómatas, nos propusimos añadirle "pereza" al proceso completo que va desde dos expresiones regulares hasta un resultado de equivalencia o no, con su consiguiente contraejemplo que demuestre la no equivalencia, si procede.

En este apartado, explicaremos las decisiones que se han tomado a la hora de mezclar algoritmos (y cómo se ha hecho) para evitar la construcción completa de todos los autómatas cuando no sea necesario. En el caso de que las dos expresiones regulares introducidas sean equivalentes, la creación de los autómatas completos se hace imprescindible para determinar su equivalencia.

La idea para añadirle pereza a los algoritmos se basa en ir creando los autómatas según el algoritmo de Hopcroft-karp los vaya utilizando (cuando se pueda) y siempre que el proceso, por el momento, no haya encontrado ninguna diferencia entre ambos autómatas. Para ello, basta formular, cuando sea posible, versiones "perezosas" de las funciones de transición de los autómatas involucrados. De esta forma, el apartado 3.1 describe algunas consideraciones preliminares. El apartado 3.2 describe cómo introducir pereza en el método de conversión de expresiones regulares en AFD basado en derivadas. El apartado 3.3 describe cómo introducir pereza en el método de determinación por subconjuntos. El apartado 3.4 describe comprobadores de equivalencia basados en, respectivamente, el método de Thompson, de seguidores y de Berri-Sethi. El apartado 3.5 describe, por último, un comprobador de equivalencia basado en el método de las derivadas parciales.

3.1 – Consideraciones preliminares

En la realización de la comprobación de la equivalencia se tendrán en cuenta los dos aspectos siguientes:

- Los dos autómatas sobre los que opera el método de Hopcroft-Karp deben tener el mismo alfabeto de entrada. Por tanto, las expresiones regulares deberán contener exactamente los mismos símbolos. De lo contrario, se resolverán como *no equivalentes* sin realizar análisis adicionales.
- El método de Hopcroft-Karp debe extenderse para devolver, en caso de no equivalencia, un *contraejemplo*: es decir, una cadena *w* que sea aceptada por uno de los autómatas pero no por el otro. Puede devolver, además, el número de autómata que acepta la cadena.

La extensión del algoritmo de Hopcroft-Karp para que proporcione, en caso de no equivalencia, un contraejemplo, es sencilla: basta añadir a los objetivos mantenidos por el algoritmo la cadena que ha llevado a plantear dichos objetivos. Con ello, los objetivos pasan a ser tripletas de la forma (w, q, q') donde: q y q' son los estados que se suponen equivalentes; y w es la cadena que ha conducido a plantear dicho objetivo. Con estas consideraciones, resulta el Algoritmo 5 (se remarcan los cambios frente al presentado en el apartado 2.4):

```
Para cada estado q de M0 y M1 se fija la clase de equivalencia de q como {q}
Sean q0 y q'0 los estados iniciales de, respectivamente, M0 y M1:
   Si uno de ellos es final y el otro no,
      devolver ("no equivalentes",λ, n° autómata que acepta la cadena)
   Registrar (λ, q0,q'0) como objetivo a explorar
   Fijar la clase de equivalencia de q0 y q'0 a {q0, q'0}
   Mientras haya objetivos a explorar:
      Extraer uno de estos objetivos, (w,q,q') del registro
      Para cada símbolo a:
         Si δ(q,a) es final y δ'(q',a) es no final, o a la inversa, devolver
            ("no equivalentes", w, n° de autómata que acepta la cadena)
         En otro caso
            Sean C_δ(q,a) y C_δ'(q',a) las clases de equivalencia de,
            respectivamente, δ(q,a) y δ'(q',a)
            Si C_δ(q,a) ≠ C_δ'(q',a)
              Registrar (wa, δ(q,a), δ'(q',a)) como objetivo a explorar
              Para cada estado q'' en C_δ(q,a) ∪ C_δ'(q',a)
                Fijar C_δ(q,a) ∪ C_δ'(q',a) como la clase de equivalencia de q''
    Devolver "equivalentes"
```

Algoritmo 5: Hopcroft-karp que devuelve contraejemplo

3.2 – Comprobador de equivalencia basado en derivadas

En este comprobador, los AFD de entrada al algoritmo de Hopcroft-Karp se obtienen mediante el método de las derivadas. Dado que, en el método de las derivadas, la expresión regular asociada con cada estado proporciona información suficiente para determinar los estados siguientes, es directo proporcionar una versión perezosa de la función de transición de estos autómatas. Para ello, dada la función de transición actual δ, un estado q, y un símbolo por el que se desea transitar a, para determinar el estado al que se transita se sigue el método indicado en el Algoritmo 6.

```
Si δ (q, a) no está definida, entonces:
  Sea α la expresión regular asociada a q
   Buscar un estado q' que tenga asociada la expresión δ_a(α)
   Si q' no existe
    q' = nuevo Estado(δ_a(α))
   fin si
   Fijar δ (q, a) a q'
fin si
Devolver δ (q, a)
```

Algoritmo 6: cálculo perezoso de las transiciones en el método de las derivadas

Tal y como indica el Algoritmo 6, en caso de que no se conozca el estado al que se transita:

- Se busca, primeramente, si existe algún estado que tenga asociada la derivada de la expresión para el estado actual (aquí se utilizan normalizaciones de las derivadas, como el criterio de ACI-congruencia).
- Si dicho estado existe, se actualiza la función de transición en consecuencia. Si no, se crea un nuevo estado que tiene asociada la correspondiente derivada y se actualiza la función para que se transite al nuevo estado.

Por tanto, dicha función se irá completando incrementalmente conforme se requiera desde el algoritmo de Hopcroft-Karp.

3.3 – Construcción por subconjuntos perezosa

El resto de los métodos de conversión de expresiones regulares en autómatas estudiados producen AFN. Por tanto, es necesario determinar dichos autómatas utilizando una construcción por subconjuntos. Como ocurre con el algoritmo de las derivadas, los estados generados por estas construcciones tienen toda la información necesaria (conjunto de estados del AFN) para obtener los estados a los que se transita. Por tanto, también en este caso es directo proporcionar versiones "perezosas" de las funciones de transición.

Supongamos que δ_{AFN} es la función de transición del AFN. Supongamos, además, que el AFN no tiene λ-transiciones. Consideremos la función de transición actual δ, un estado *q*, y un símbolo *a* por el que se desea transitar. La determinación del estado siguiente se lleva a cabo como se indica en el Algoritmo 7:

```
Si δ (q, a) no está definida, entonces:
  Sea R el conjunto de estados asociado a q
     R' = ∪ q'∈R δAFN(q', a)
     Buscar un estado q' que tenga asociado el conjunto R'
     Si q' no existe
       q' = nuevo Estado(R')
     fin si
     Fijar δ (q, a) a q'
fin si
Devolver δ (q, a)
```

Algoritmo 7: Cálculo perezoso de las transiciones en el método de los subconjuntos para AFNs sin λ-transiciones

En caso de que el AFN pueda tener λ-transiciones, la adaptación del método es directa (basta aplicar el LAMBDA-CIERRE). El método se describe en el Algoritmo 8.

```
Si δ (q, a) no está definida, entonces:
  Sea R el conjunto de estados asociado a q
     R' = LAMBDA-CIERRE(∪ q'∈R δAFN(q', a))
     Buscar un estado q' que tenga asociado el conjunto R'
     Si q' no existe
       q' = nuevo Estado(R')
     fin si
     Fijar δ (q, a) a q'
fin si
Devolver δ (q, a)
```

Algoritmo 8: Cálculo perezoso de las transiciones en el método de los subconjuntos para AFNs con λ-transiciones

3.4 – Comprobadores de equivalencia basados en los métodos de Thompson, Seguidores y Berry-Sethi

En los métodos de Thompson, Seguidores y Berry-Sethi, la creación de estados se basa en propiedades globales de la expresión regular. Por tanto, no es sencillo idear mecanismos que construcción perezosa de los correspondientes AFN. Es por ello que, en todos estos casos, los comprobadores de equivalencia propuestos operan en dos fases:

- En la primera aplican el correspondiente método para transformar la expresión regular en un AFN equivalente.
- En la segunda utilizan el método de Hopcroft-Karp junto con la versión perezosa adecuada de la construcción por subconjuntos (construcción para AFN con λ-transiciones en el caso de Thompson y construcción para AFN sin λ-transiciones en los otros casos).

3.5 – Comprobadores de equivalencia basado en el método de las derivadas parciales

En el caso del método de las derivadas parciales es posible proporcionar también una versión perezosa de la función de transición de AFN producido por el método. Supongamos que deseamos determinar a qué estados se transita desde el estado *q* mediante el símbolo *a*. Supongamos, asimismo, que la función de transición del AFN actual es δ_{AFN}. Puede aplicarse la estrategia descrita en el Algoritmo 9.

```
si δAFN(q,a) no está definida, entonces:
  Sea α la expresión regular asociada a q
  Fijar δAFN(q,a) a ∅
  Para cada β en ∂a(α):
    Buscar un estado q' que tenga asociada la derivada parcial β
    Si q' no existe
       q' = nuevo Estado(β)
    fin si
    Añadir q' a δAFN(q,a)
  fin para
 fin si
 Devolver δAFN(q,a)
```

Algoritmo 9: Cálculo perezoso de las transiciones en el método de las derivadas parciales

De esta forma, en el comprobador resultante se desarrollarán, perezosamente, no solo los AFD asociados con las expresiones regulares, sino también los correspondientes AFN obtenidos mediante el método de las derivadas parciales. Esto confiere al método un gran atractivo cuando las expresiones de entrada no son equivalentes [15].

4. Diseño e implementación de un sistema para la comprobación de la equivalencia de especificaciones basadas en definiciones regulares

En este capítulo nos centraremos en la herramienta de comprobación de la equivalencia de especificaciones basadas en expresiones regulares desarrollada en este TFG, la que verá y utilizará el usuario para realizar las comprobaciones que desee. En el apartado 4.1 describimos la arquitectura del sistema. En el apartado 4.2 describimos el lenguaje en el que se proporcionan las especificaciones basadas en expresiones regulares. En el apartado 4.3 describimos el proceso de carga de las especificaciones en la herramienta. En el apartado 4.4 describimos los mecanismos de comprobación proporcionados por la herramienta. En el apartado 4.5 describimos la interfaz de usuario.

El código de esta herramienta está disponible en nuestro repositorio de GitHub: https://github.com/7216nat/TFG_Comprobador-de-equivalencia-entre-especificaciones-basadas-en-expresiones-regulares

4.1 – Arquitectura del sistema

Se describe la arquitectura del sistema con la ayuda del diagrama de clases UML[6] de la Figura 12 utilizando la herramienta de *open source* Modelio[7].

Como se puede observar, en nuestro programa hemos utilizado el patrón de diseño Modelo-Vista-Controlador (MVC) descrito más adelante en este mismo apartado.

El MVC, explicado por E. Gamma, R. Helm, R, Johnson y J. Vlissides [12], es un patrón de arquitectura software que permite separar la parte visual de la parte lógica de una aplicación. Esto significa que la interfaz gráfica de una aplicación no tiene ningún tipo de contacto directo con la parte del modelo de datos y viceversa. En nuestro programa:

- El Modelo está compuesto por los tipos de expresión regular, el autómata y la lógica que se utiliza para su creación o tratamiento que permite llevar a cabo la comparación entre dos autómatas. Es en esta capa donde se desarrolla la operatividad de la aplicación.
- La Vista corresponde a toda la parte gráfica de la aplicación. Esta parte constituye únicamente la interfaz gráfica que permite la interacción del usuario con la aplicación.
- Por último, el Controlador permite distribuir las peticiones que la Vista envía al Modelo, por lo que es una capa intermedia entre ambas.

[6] UML, Lenguaje Unificado de Modelado, es un lenguaje gráfico que permite visualizar, especificar y documentar un sistema: https://www.uml.org/.

[7] https://www.modelio.org/

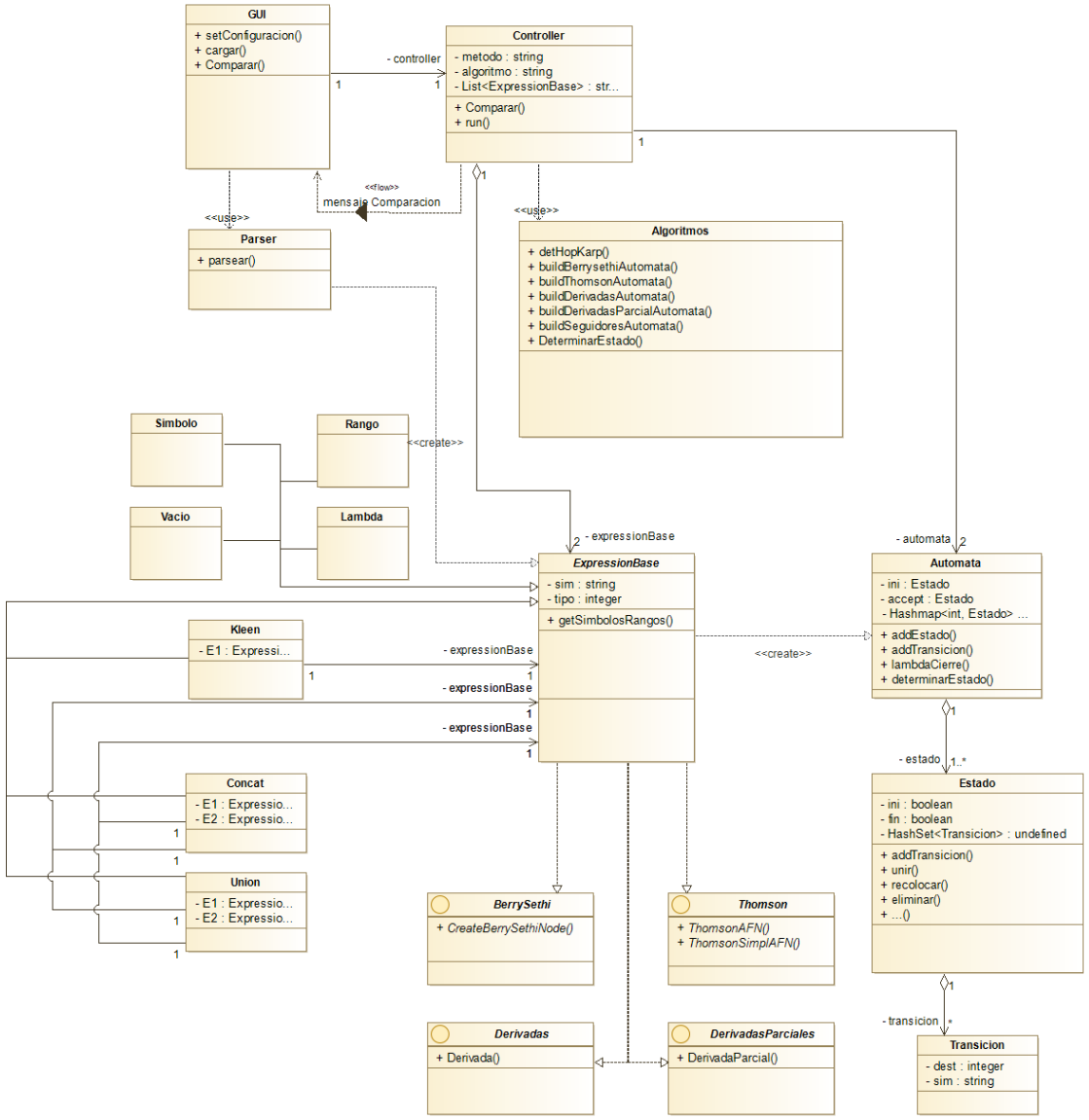

Figura 12: arquitectura del sistema

4.1.1 – Implementación del MVC en el proyecto

- **Acciones Vista-Controlador (las clases `GUI` y `Controlador`)**
  1. Carga de las especificaciones: mediante el buscador de archivo proporcionado por el marco Swing[8] de Java se selecciona el archivo del que se obtienen las listas con las especificaciones en forma de expresiones regulares. Éstas son mostradas al usuario y seleccionables para realizar las peticiones correspondientes.
  2. Realizar la comparación: el Controlador recoge las configuraciones seleccionadas y los conjuntos de expresiones regulares a comparar en la Vista para poder invocar los métodos adecuados. Finalmente devuelve el resultado en la consola de Vista.

[8] https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/package-summary.html

- **Acciones Controlador-Modelo**
  1. Carga de las especificaciones: dado el archivo, el Modelo realiza primero un análisis léxico y luego un análisis sintáctico combinado con procesamiento dirigido por sintaxis (detallado todo ello más adelante) de forma que, al final, cada especificación tenga su expresión regular representada por un árbol de sintaxis abstracta.
  2. Realizar la comparación: según las configuraciones (el algoritmo y el método elegidos) ejecuta las subrutinas correspondientes para comparar dos expresiones regulares, devolviendo un mensaje (equivalentes, posiblemente equivalentes o no equivalentes). En caso negativo se devuelve también un ejemplo de discrepancia entre ellas.

- **Modelo**

Está compuesto por el resto de las clases, brevemente descritas a continuación:

1. La clase abstracta `ExpresionBase` es la clase base para los distintos tipos de nodos del árbol de sintaxis abstracta de la expresión regular. Todas las clases derivadas coinciden en el nombre del constructor correspondiente en el formalismo de las expresiones regulares, excepto la clase `Rango`, que se utiliza para representar conjuntos de símbolos consecutivos (el alfabeto de las expresiones regulares utilizadas es el juego de caracteres UNICODE, y esta clase permite representar subconjuntos significativos, por ejemplo: *letra minúscula*, *dígito* …).
2. Clase `Parser`: para poder procesar los archivos de entrada y generar todos los árboles de sintaxis abstracta (AST) que representan las expresiones regulares, es necesario construir un *parser*, que se encargará de comprobar los errores léxicos o sintácticos y de la traducción del texto a ASTs. El *parser* actúa de la siguiente manera, que también caracteriza su estructura:
   I. Análisis léxico: se ha utilizado la herramienta JFlex[9] para construir un analizador léxico de acuerdo con las reglas de especificación de lenguaje (véase la sección 4.2). Dicho código fuente se puede consultar en el repositorio de Github.
   II. Análisis sintáctico: para la construcción se ha utilizado la herramienta CUP[10]. Se han definido las reglas para la generación automática de un analizador sintáctico ascendente LALR(1)[11], que también se puede consultar en el repositorio de Github.
   III. Procesamiento dirigido por la sintaxis: el objetivo final del *parser* es generar un AST cuyos nodos son los diferentes tipos básicos de expresión regular. Un método sistemático de conseguirlo es modificar el analizador sintáctico para que pueda soportar una *gramática de atributos* s-*atribuida*[12], que puede asociar un valor

[9] https://jflex.de/

[10] https://www.cs.princeton.edu/~appel/modern/java/CUP/

[11] Un analizador LALR(1) es un analizador ascendente que lee símbolos de la entrada de izquierda a derecha y construyen el árbol sintáctico en sentido ascendente, siguiendo una derivación por la derecha en sentido inverso.

[12] Las *gramáticas de atributos* son un formalismo propuesto por Knuth [13] para especificar la semántica de los lenguajes incontextuales. La semántica de una sentencia según Knuth es la asociación de otra sentencia en un lenguaje objeto (su significado). Para determinar la semántica de una construcción sintáctica puede usarse (i) la semántica de sus constituyentes (*atributos sintetizados*); (ii) información de contexto adicional (*atributos heredados*). La gramática es *s-atribuida* cuando no utiliza información de contexto (es decir, únicamente involucra atributos sintetizados) [10].

semántico (en este caso un nodo de expresión regular) a cada una de las producciones o reglas. De esta forma, después del análisis se devuelve un AST que representa la expresión regular. Se ha utilizado la misma herramienta que el analizador sintáctico, CUP, para la implementación automática de la gramática s-atribuida. Las especificaciones se pueden consultar al detalle en nuestro repositorio Github.

3. `Automata`, `Estado`, `Transicion`: son clases de estructuras de datos que representan un autómata, estado y transición respectivamente.
4. Las interfaces `Thompson`, `BerrySethi`, `Derivadas` y `DerivadasParciales` contienen métodos abstractos que transforman la expresión regular a una estructura de datos que denota el autómata.
5. La clase `Algoritmos` contiene todos los algoritmos que se han utilizado en la herramienta para la creación y determinación de autómatas y la comprobación de equivalencia entre ellos. Todos los algoritmos reciben dos instancias de la clase `Automata` (entre otras cosas) y devuelven como resultado un mensaje indicando el resultado de la comparación.

4.2 – Lenguaje de especificación

Para la entrada del lenguaje en el sistema se utiliza un formato sencillo para describir especificaciones basadas en expresiones regulares. Dicho lenguaje, aparte del formalismo básico de las expresiones regulares, incorpora nuevos operadores (traducibles directamente a los básicos), rangos, y un mecanismo de abstracción basado en *definiciones regulares* [9] que permite declarar expresiones más simples y utilizarlas más adelante en la formación de expresiones más complejas. De esta forma:

- La especificación está definida por *auxiliares* y *declaraciones*, y éstas últimas son las que aparecerán en el programa para realizar la comparación. Los auxiliares solo se utilizan para facilitar la escritura de las especificaciones.
- Cada definición, ya sea de un auxiliar o de una declaración, se escribirá en una línea independiente.
- La parte de auxiliares se escribe siempre primero, y debe detallarse como una lista de definiciones auxiliares, cada una de ellas con el siguiente formato:

  aux <clave> = expresión_regular

  donde:
  - **aux** es una palabra reservada.
  - *clave* es el nombre con el que el programa almacenará la expresión regular. Se escribe siempre entre símbolos de menor y mayor: “< *clave* >”. La clave puede contener cualquier carácter ASCII excepto el símbolo “>”, que dará por concluida la clave.
  - *expresión_regular* es la expresión regular que queremos que defina el auxiliar.

- La parte de declaraciones va después de la parte de auxiliares, y consta de una lista de definiciones, cada una con el formato:

  def <clave> = expresión_regular

  donde:
  - **def** es una palabra reservada.
  - *clave* y *expresión_regular* son como en la definición de auxiliares.
- Las expresiones regulares se describen como sigue (en la descripción, α y β son expresiones regulares, y **[**α**]** y **[**β**]** son las correspondientes representaciones en la notación introducida):
  - ∅ se escribe como %.
  - λ se escribe como &.
  - La expresión *c* asociada a un carácter que *no* es un metacarácter.
  - [a - b] representa un rango, donde *a* es el límite inferior del rango y *b* es el límite superior, ambos incluidos en el rango. Siempre debe cumplirse que a < b. Puede representarse la unión de varios rangos separándolos mediante comas: [a - z, 0 - 9].
  - α+β se escribe como **[**α**]**|**[**β**]**.
  - αβ se escribe como **[**α**][**β**]**.
  - α* se escribe como **[**α**]***.
  - **[**α**]**+ denota la expresión αα*.
  - **[**α**]**? denota la expresión (α | λ)
  - <*clave*>: si quiere utilizarse una definición auxiliar (previamente definida), debe escribirse la clave entre los símbolos menor y mayor. No pueden referenciarse claves de declaraciones, solo de auxiliares.

  El operador | tiene la menor prioridad, seguido de la concatenación, seguida de los operadores unarios. Se pueden usar paréntesis para cambiar dichas prioridades. Asimismo, cuando a la expresión regular se le quiera añadir, de forma literal, uno de los metacaracteres (como el paréntesis de apertura o el asterisco) o alguna de las palabras reservadas "aux" o "def", tendrá que escribirse precedido de una barra de escape, es decir: "\(", "*", "\aux", "\def". Aunque en el caso de las palabras "aux" y "def", la barra de escape puede escribirse en cualquier punto intermedio de la palabra, nunca al final.

  A continuación se listan todos los metacaracteres que necesitan una barra de escape para introducirlos literalmente en el lenguaje: '&', '%', "def", "aux", '=', '<', '>', '|', '*', '+', '?', '(', ')', '[', ']', '-', ',' y '\'.
- Cuando quieran añadirse blancos, retornos de carro, saltos de línea o tabulaciones, deben escribirse como "\b", "\r", "\n" y "\t", respectivamente. Si no, serán ignorados o se interpretará como una nueva definición si se cambia de línea.

A continuación, se muestra un ejemplo de especificación aceptada:

```
aux <digitoNoCero> = [1-9]
aux <letra> = [a-z, A-Z]
def <num> = <digitoNoCero> (<digitoNoCero> | 0)*
def <palabra> = <letra> +
```

En el capítulo 5 se incluyen otras especificaciones que pueden servir como ejemplos de entradas válidas.

4.3 – Proceso de carga de las especificaciones

Las especificaciones se cargan a través de un archivo de texto simple (formato ".txt") con el formato de escritura que se ha detallado en el apartado 4.2. Este archivo puede estar almacenado en cualquier unidad de almacenamiento del ordenador, dado que para la carga se abrirá una ventana de selección de archivo con navegación por carpetas. Esta exploración empezará, por defecto, en la carpeta *Documents*.

En un primer paso, se hace un análisis léxico para comprobar que no hay errores de tipo léxico en el archivo.

Después, se hace un análisis sintáctico donde se comprueban posibles errores sintácticos (en cuyo cayo no puede cargarse el lenguaje y se muestra un error en la aplicación), y se construyen los árboles de las distintas expresiones regulares y un *diccionario* donde éstas se asocian a los nombres, o claves, que se hayan indicado en el archivo.

4.4 – Métodos de comprobación

En el siguiente apartado, el 4.5, se muestran las diferentes opciones que tenemos para la selección de algoritmos, aquí las listamos y describimos la secuencia de algoritmos que se siguen:

- Thompson:
  - Algoritmo de Thompson.
  - Determinación por subconjuntos con λ-transiciones (versión perezosa) + comprobación de equivalencia por algoritmo de Hopcroft-Karp.
- Seguidores:
  - Algoritmo de seguidores.
  - Determinación por subconjuntos sin λ-transiciones (versión perezosa) + comprobación de equivalencia por algoritmo de Hopcroft-Karp.
- Derivadas:
  - Algoritmo de derivadas (versión perezosa) + comprobación de equivalencia por algoritmo de Hopcroft-Karp.
- Derivadas parciales:
  - Algoritmo de las derivadas parciales (versión perezosa) + determinación por subconjuntos sin λ-transiciones (versión perezosa) + algoritmo de Hopcroft-Karp.
- Berry-Sethi:
  - Algoritmo de Berry-Sethi.
  - Determinación por subconjuntos sin λ-transiciones + comprobación de equivalencia por algoritmo de Hopcroft-Karp.

La herramienta también dispone de varios métodos de comparación de las especificaciones a evaluar:

- *Seleccionados*: se hará la unión de todas las definiciones que se hayan seleccionado en cada especificación y se compararán entre ellas. Así podremos comparar manualmente la equivalencia entre expresiones concretas.
  La no equivalencia es *segura*, pero si nos devuelve *posiblemente equivalentes*, no es del todo seguro que las especificaciones sean equivalentes entre sí, debido a que la unión de varias definiciones puede ser igual en ambos lenguajes, pero discrepar en las definiciones individuales.
- *Uno a uno*: independientemente de las que marquemos en la lista, se comparará cada definición de la lista de la primera especificación con las definiciones de la lista de la segunda especificación, estableciendo relaciones uno a uno.
  Este método devuelve la lista de relaciones uno a uno emparejadas. En este caso, la resolución de equivalencia es segura. También devuelve la lista de expresiones regulares de ambas especificaciones que no han podido emparejarse con ninguna de la otra.
- *Todos*: se hará la unión de todas las expresiones regulares de cada especificación, independientemente de las que tengamos marcadas en la pantalla, y se hará la comprobación de equivalencia con el resultado.
  Devolverá si son *posiblemente equivalentes* o *no equivalentes* y, en el segundo caso, también una cadena que será aceptada por una especificación y rechazada por la otra.
  De nuevo, la no equivalencia entre especificaciones es segura, pero la "posible equivalencia" entre ellos no. Esto es debido a que, si hacemos la unión de todas las expresiones de cada lenguaje y luego comparamos el resultado, éstas pueden ser equivalentes aunque existan discrepancias entre las definiciones individuales.

4.5 – Interfaz de usuario

Esta es una herramienta orientada al aprendizaje, por lo que hemos optado por una interfaz sencilla y adecuada al nivel en el que nos encontramos (ya centrados en una asignatura en la que se enseñen expresiones regulares y se conozca el significado de los conceptos básicos de la teoría de lenguajes, expresiones regulares, etc.). Se tendrán todas las funciones a mano desde un primer momento.

Los detalles de las diferentes partes de la interfaz son como sigue (Figura 13):

- Cargar lenguaje: se abrirá una ventana de navegación por directorios donde deberemos buscar el fichero con formato ".txt" en el que se encuentre la especificación cuya equivalencia con otra queramos estudiar. En caso de que la especificación esté bien descrita, sus definiciones se mostrarán en las listas inferiores.
- Listas de lenguaje: aquí podremos seleccionar una o varias declaraciones (las definidas como auxiliares no aparecerán). Por defecto, aparecen por nombre, pero podemos cambiar esto pulsando el botón "expresiones".

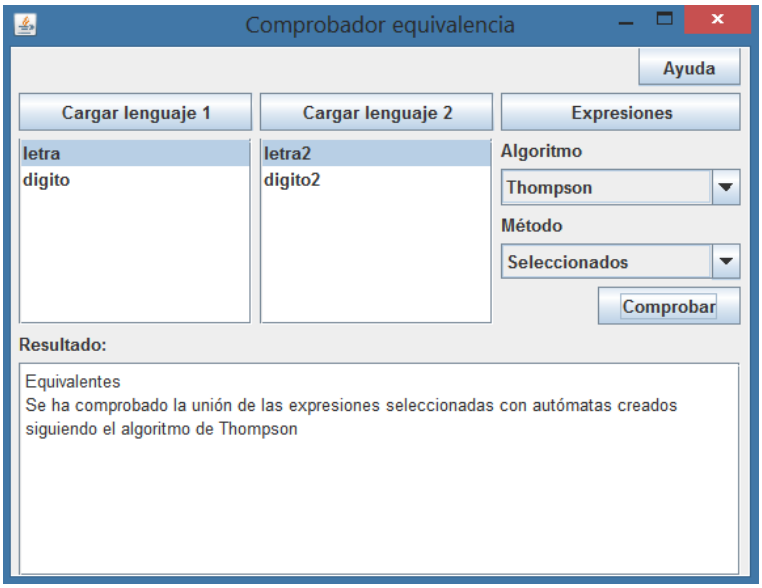

Figura 13: pantalla principal de la herramienta

- Expresiones: cambiará la vista de las listas entre nombres y las expresiones regulares a las que representan.
- Selector "Algoritmo": podremos seleccionar el algoritmo de creación de autómatas finitos que deseemos (Figura 14).
    - Thompson
    - Seguidores
    - Derivadas
    - Derivadas parciales
    - Berry-Sethi
- Selector método: podremos seleccionar el método de entrada, estos son (Figura 15):
    - Seleccionados
    - Uno a uno
    - Todos

Figura 14: selector del algoritmo

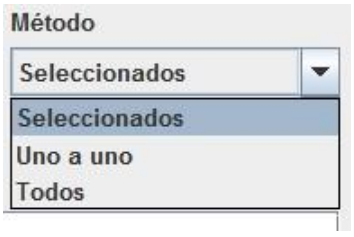

Figura 15: selector del método de entrada

- Comprobación: pone en funcionamiento la comprobación de equivalencia solicitada (con el método y el algoritmo que hayamos escogido). Muestra el resultado en la pantalla de resultados.

- Panel de resultados: aquí aparecerán tanto los resultados de las comprobaciones como los errores en caso de que se haya intentado cargar un lenguaje especificado de forma incorrecta.
  Cuando muestre la solución, y para mantener una visión completa del resultado, mostrará, al final, el algoritmo de creación de autómatas finitos que se ha utilizado y el método de entrada.
- Ayuda: muestra una ventana de ayuda con información de (Figura 16):
  - Lenguaje: tipo de archivo y cómo especificar correctamente el lenguaje.
  - Programa: instrucciones para usar correctamente el programa.
  - Info: aporta información de cómo funciona internamente el programa, es decir, indica la secuencia de algoritmos que se sigue con cada opción de algoritmos y qué expresiones regulares y de qué forma se toman según el método seleccionado.

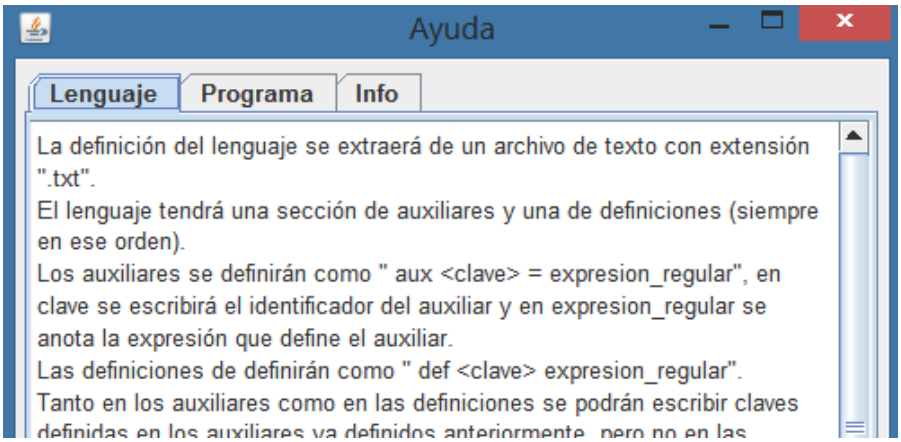

Figura 16: ventana de ayuda

5. Evaluación preliminar

Ya con la herramienta completa, hemos seguido un proceso de pruebas gracias a una pequeña batería de especificaciones proporcionada por el director de este TFG.

Partimos de una especificación realizada por el profesor y otras 10 especificaciones con las que compararla (estas especificaciones han sido extraídas aleatoriamente por el profesor de distintas entregas realizadas por estudiantes de la asignatura de "Procesadores de Lenguajes" en sus trabajos prácticos).

Para cada especificación realizada por estudiantes:

- Se han realizado las pruebas mediante el método de emparejamiento uno a uno y todos.
- Para aquellas que salieran desemparejadas, las hemos evaluado comparando directamente aquellas declaraciones que deberían representar la misma expresión regular, indicando claramente las dos definiciones comparadas.

Las pruebas se han hecho con el algoritmo de Thompson y se han repetido con todos los demás algoritmos. En caso de que alguna de las otras cuatro secuencias de algoritmos diera otro resultado, se especificará cuál y el resultado obtenido por esta.

A la hora de exponer los resultados, con *lenguaje1* nos referiremos a la especificación proporcionada por el profesor, la correcta, y con *lenguaje2* nos referiremos siempre a aquella con la que la estemos comparando.

También se proporcionan los resultados de la corrección del profesor, para comparar.

5.1 – Especificación del profesor

```
aux <Letra> =  [a-z] | [A-Z]
aux <DPos> = [1-9]
aux <Digito> = <DPos> | 0
aux <Signo> =(\+|\-)
aux <PEnt> = (<DPos> <Digito>* | 0)
aux <PDec> = . (<Digito>* <DPos> | 0)
aux <PExp> = (e|E) <Signo>?  <PEnt>
def <Iden> = <Letra> (<Letra> | <Digito> | _)*
def <LEnt> = <Signo>? <PEnt>
def <LReal> = <Signo>? <PEnt> (<PDec> | <PExp> | <PDec> <PExp>)
```

5.2 – Especificación de prueba 1

```
aux <DigitoPos> =  [1-9]
aux <Digito> = (0 | [1-9])
aux <Ent> = <DigitoPos> <Digito>*
aux <PDec> = . <Digito>* <DigitoPos>
aux <PExp> = (e | E) (\-)? <DigitoPos> <Digito>*
aux <letra> = [a-z, A-Z]
def <EntNum> = ( \- | \+)? ( <Ent> | 0)
def <RealNum> = (\- | \+)? (<Ent> | 0) (<PDec> | <PExp> | <PDec> <PExp>)
def <Variable> = <letra> ( <letra> | <Digito> | \_ )*
```

- Uno a uno:
    - Emparejadas: Iden–Variable y LEnt–EntNum.
    - Desemparejadas: LReal y RealNum.
- Todos: cadena 0e0: aceptada por lenguaje1 y rechazada por lenguaje2.
- Seleccionados (LReal y RealNum):
    - LReal acepta 0e0 y RealNum no.
- Corrección del profesor:
    - No se admite un .0 como parte decimal. No se admite un signo + en la parte exponencial.
- Comentarios:
    - La herramienta ha descubierto una discrepancia no señalada inicialmente por el profesor, pero que también existe entre ambas especificaciones.

5.3 – Especificación de prueba 2

```
aux <letra> = [a-z, A-Z]
aux <dig> = [0-9]
aux <digPos> = [1-9]
aux <nat> = (<digPos> <dig>*) | 0
aux <pExp> = (e|E) (\-)? <nat>
aux <pDec> = . ((<dig>* <digPos>)|0)
aux <enteroAux> = (\+ | \-)? <nat>
def <variable> = <letra> (<letra> | <dig> | _)*
def <entero> = <enteroAux>
def <num_real> = <enteroAux> (<pDec> | <pExp>| <pDec> <pExp>)
```

- Uno a uno:
    - Empareja Iden–variable y LEnt–entero.
    - LReal y num_real no empareja.
- Todos:
    - Cadena intermedia 0e+ posible en lenguaje1 e imposible en lenguaje2 (esto es debido a que las implementaciones basadas en los métodos de Thompson, seguidores y Berry-Sethi realizadas en la herramienta permiten detener el proceso cuando se descubre que en el estado actual de uno de los autómatas existe una transición a través de un determinado símbolo que, en el otro, no existe).
    - Derivadas y derivadas parciales: Cadena 0e+0 aceptada por lenguaje 1, no aceptada por lenguaje 2.
- Seleccionados (LReal y num_real) :
        - Cadena intermedia 0e+ posible en lenguaje1 e imposible en lenguaje2.
        - Derivadas y derivadas parciales: Cadena 0e+0 aceptada por lenguaje 1, no aceptada por lenguaje 2.
- Corrección del profesor:
    - No se admite un signo + en la parte exponencial.
- Comentarios:
    - En este caso, el diagnóstico proporcionado por la herramienta coindice con el proporcionado por el profesor.

5.4 – Especificación de prueba 3

```
aux <letra> = [a-z, A-Z]
aux <digPos> = [1-9]
aux <dig> = (<digPos> | 0)
aux <ParteEnt> = (<digPos>) (<dig>)*
aux <ParteReal> = (<dig>)* (<digPos>)
aux <NumEntAux> = (\+ | \-) ? (<ParteEnt>)
def <Var> = <letra> (<letra> | <dig> | _ )*
def <NumEnt> = <NumEntAux>
def <NumReal> = (\+ | \-) ? (<ParteEnt>) ( \. (<ParteReal>)) ? (( E | e )
(<NumEntAux> | 0)) ?
```

- Uno a uno:
  - Empareja Iden–Var.
  - No empareja LEnt, LReal, NumEnt, NumReal.
- Todos:
  - Cadena 0: aceptada por lenguaje1, rechazada por lenguaje 2.
- Seleccionados:
  - LEnt y NumEnt: Cadena 0: aceptada por lenguaje 1, rechazada por lenguaje2.
  - LReal y NumReal:
    - Cadena intermedia 0: posible en lenguaje1, imposible en lenguaje2.
    - Derivadas y derivadas parciales: Cadena 1: rechazada por lenguaje1, aceptada por lenguaje2.
- Corrección del profesor:
  - Los números reales deben tener al menos una parte decimal y/o una exponencial (la definición dada admite números reales que no tienen ninguna de esas partes). No se admite .0 como parte decimal.
- Comentarios:
  - La herramienta ha descubierto una discrepancia no señalada inicialmente por el profesor, pero que también existe entre ambas especificaciones.

5.5 – Especificación de prueba 4

```
aux <letra> = [a-z, A-Z]
aux <digito> = [0-9]
aux <enteros> = ([1-9] <digito>* | 0)
aux <decimales> = (<digito>* [1-9] | 0)
aux <Sum> = \+
aux <Res> = \-
aux <NIntAux> = (<Sum> | <Res>)? <enteros>
def <Var> = <letra> (<letra> | <digito> | _) *
def <NInt> = <NIntAux>
def <NReal> = <NIntAux>. <decimales> ((e | E) <enteros>)?
```

- Uno a uno:
  - Empareja Iden–Var y LEnt–NInt.
  - LReal y `NReal`: LReal acepta 0e0 y `NReal` no.
- Todos:
  - Cadena intermedia 0e: posible en lenguaje1, imposible en lenguaje2.
  - Derivadas y derivadas parciales: Cadena 0e0: aceptada por lenguaje1, rechazada por lenguaje2.
- Seleccionados (LReal y NReal):
  - Cadena intermedia 0e: posible en lenguaje1, imposible en lenguaje2.

    - Derivadas y derivadas parciales: Cadena 0e0: aceptada por lenguaje1, rechazada por lenguaje2.
- Corrección del profesor:
    - La parte exponencial puede llevar un signo (no se está permitiendo).
- Comentarios:
    - Claramente la herramienta ha sido aquí más precisa que el profesor, al descubrir un defecto fundamental en la especificación de los estudiantes: se está obligando a que aparezca siempre una parte decimal en los literales reales.

5.6 – Especificación de prueba 5

```
aux <letra> = [a-z, A-Z]
aux <digito> = [0-9]
aux <dig_nozero> = [1-9]
aux <decimal> = . <digito>* <dig_nozero>
aux <enteroAux> = (\+|\-)? <dig_nozero> <digito>*
aux <exponencial> = (e|E) <enteroAux>
def <entero> = <enteroAux>
def <real> = <enteroAux> (<decimal> | <exponencial> | <decimal> <exponencial>)
def <variable> = <letra> (<letra> | <digito> | _)*
```

- Todos:
    - Cadena 0: aceptada por lenguaje1, rechazada por lenguaje2.
- Uno a uno:
    - Emparejados: Iden–variable.
    - Desemparejados: LEnt, LReal, entero, real.
- Seleccionados:
    - LEnt – entero: cadena 0 aceptada por lenguaje1, rechazada por lenguaje2.
    - LReal – real:
        - Cadena intermedia 0: posible en lenguaje1, imposible en lenguaje2.
        - Derivadas y derivadas parciales: cadena 0e0 aceptada por lenguaje1, rechazada por lenguaje2.
- Corrección:
    - No se permite reconocer 0, +0, -0 como un número entero. No se permite .0 como parte decimal. No se permite 0, +0, -0 como parte exponencial.
- Comentarios:
    - En este caso, la herramienta permite descubrir aspectos similares a los indicados por el profesor.

5.7 – Especificación de prueba 6

```
aux <letra> = [a-z, A-Z]
aux <dig> = [0-9]
aux <digPos> = [1-9]
aux <PDec> = . (<dig>*)? <digPos>
aux <PExp> = (e|E) (\+|\-)? <digPos> (<dig>*)?
def <Variable> = <letra> (<letra> | <dig> | _)*
def <NumEnt> = (\+|\-)? <digPos> (<dig>*)?
def <NumReal> = (\+|\-)? <digPos> (<dig>*)? <PDec> (<PExp>)?
```

- Uno a uno:
    - Emparejadas: Iden–Variable.
    - Desemparejadas: LEnt, LReal, NumEnt, NumReal.

- Todos:
  - Cadena 0: aceptada por lenguaje 1, rechazada por lenguaje2.
- Seleccionados:
  - LEnt y NumEnt: Cadena 0: aceptada por lenguaje 1, rechazada por lenguaje2.
  - LReal y NumReal:
    - Cadena intermedia 0: posible en lenguaje1, imposible en lenguaje 2.
    - Derivadas y derivadas parciales: cadena 0e0 aceptada por lenguaje1, rechazada por lenguaje2.
- Corrección:
  - No se permite reconocer 0, +0, -0 como un número entero. No se permite .0 como parte decimal. No se permite 0, +0, -0 como parte exponencial. Se obliga a que aparezca siempre una parte decimal (esto no tiene por qué ser así).
- Comentarios:
  - La herramienta evidencia el primero de los aspectos señalados por el profesor.

5.8 – Especificación de prueba 7

```
aux <DigPos> = [1-9]
aux <Digito> = [0-9]
aux <NumPos> = <DigPos> <Digito>*
aux <Exp> =  (e|E) <Digito>
aux <Decimal> = . <Digito>* <DigPos>
aux <Letra> = [a-z,A-Z]
def <SimProp> = <Letra> (<Letra>| _| <Digito>)*
def <Int> =  ( (\+) | (\-) )? (0 | <NumPos>)
def <Real> = <Digito> (<Decimal> | <Exp> | <Decimal> <Exp>)
```

- Uno a uno:
  - Emparejadas: Iden–SimProp y LEnt–Int.
  - Desemparejadas: LReal y Real.
- Todos:
  - Cadena intermedia 0e+: posible en lenguaje1, imposible en lenguaje 2.
  - Derivadas y derivadas parciales: cadena 0.0: aceptada por lenguaje1, rechazada por lenguaje 2.
- Seleccionados (LReal y Real):
  - Cadena intermedia +: posible en lenguaje1, imposible en lenguaje2.
  - Cadena 0.0: aceptada por lenguaje1, rechazada por lenguaje2.
- Corrección:
  - No se permite .0 como parte decimal.
- Comentarios:
  - La herramienta evidencia el aspecto señalado por el profesor, así como otro obviado por el mismo (el de la no posibilidad de incluir signo en la parte exponencial).

5.9 – Especificación de prueba 8

```
aux <DigPos> = [1-9]
aux <Dig> = (<DigPos> | 0)
aux <Letra> = [a-z] | [A-Z]
aux <ParteEnt> = <DigPos> <Dig>*
aux <ParteDec> = <Dig>* <DigPos>
aux <NumEntAux> = (\+ | \-)? <ParteEnt>
def <NombreVar> = <Letra> (<Letra> | <Dig> | _)*
def <NumEnt> = <NumEntAux>
def <NumReal> = (\+ | \-)? <ParteEnt> (\. <ParteDec> (e|E) <NumEntAux> | \.
<ParteDec> | (e|E) <NumEntAux>)
```

- Uno a uno:
  - Emparejados: Iden–NombreVar.
  - Desemparejados: LEnt, LReal, NumEnt y NumReal.
- Todos:
  - Cadena 0: aceptada por lenguaje1, rechazada por lenguaje2.
- Seleccionados:
  - LEnt y NumEnt: Cadena 0: aceptada por lenguaje1, rechazada por lenguaje2.
  - LReal y NumReal:
    - Cadena intermedia 0: posible en lenguaje 1, imposible en lenguaje2.
    - Derivadas y derivadas parciales: cadena 0e0, aceptada por lenguaje1, rechazada por lenguaje2.
- Corrección del profesor:
  - No se permite 0, +0, -0 como entero. No se permite .0 como parte decimal.
- Comentarios:
  - La herramienta evidencia el primer aspecto señalado por el profesor, así como otro obviado por el mismo (el de la no posibilidad de especificar 0 en la parte exponencial).

5.10 – Especificación de prueba 9

```
aux <letra> = [a-z] | [A-Z]
aux <digitoPositivo> = [1-9]
aux <digito> = [0-9]
aux <parteDecimal> = \. <digito>* <digitoPositivo>
aux <enteroAux> = (\+ | \-)? <digitoPositivo> <digito>*
aux <parteExponencial> = (e|E) <enteroAux>
def <identificador> = <letra> (<letra> | <digito> | _)*
def <entero> = <enteroAux>
def <real> = <enteroAux> (<parteDecimal> | <parteExponencial> | <parteDecimal>
<parteExponencial>)
```

- Uno a uno:
  - Emparejados: Iden–identificador.
  - Desemparejados: LEnt, LReal, entero y real.
- Todos:
  - Cadena 0: aceptada por lenguaje1, rechazada por lenguaje2.
- Seleccionados:
  - LEnt y entero: Cadena 0: aceptada por lenguaje1, rechazada por lenguaje2.
  - LReal y real
    - Cadena intermedia 0: posible en lenguaje1, imposible en lenguaje2.
    - Derivadas y derivadas parciales: 0e0: aceptada por lenguaje1, rechazada por lenguaje2.

- Corrección del profesor:
  - No se permite 0, +0, -0 como entero. No se permite .0 como parte decimal.
- Comentarios:
  - La herramienta evidencia el primer aspecto señalado por el profesor, así como otro obviado por el mismo (el de la no posibilidad de especificar 0 en la parte exponencial).

5.11 – Especificación de prueba 10

```
aux <letra> = [a-z, A-Z]
aux <digito_pos> = [1-9]
aux <digito> = [0-9]
aux <parte_entera> = (\+|\-)? (<digito_pos> (<digito>)* | 0)
aux <parte_decimal> = . (((<digito>)* <digito_pos>) |0)
aux <parte_exp> = (E|e) <parte_entera>
def <id> = <letra> (<letra> | <digito> | _)*
def <lit_entero> = <parte_entera>
def <lit_real> = <parte_entera> (<parte_decimal> | <parte_exp> |
<parte_decimal> <parte_exp>)
```

- Uno a uno:
  - Emparejados: Iden–id, LEnt–lit_entero y LReal–lit_real.
- Todos:
  - Equivalentes.
- Corrección del profesor:
  - No ve fallos.
- Comentarios:
  - La herramienta coincide con el criterio del profesor.

5.12 – Análisis de resultados

Hemos comprobado en las especificaciones de prueba si el resultado ofrecido por la herramienta (la cadena de ejemplo en los casos de no equivalencia) eran adecuados o no, y hemos visto que, efectivamente, se cumplía su diagnóstico.

Es interesante indicar que, en muchos casos, los métodos de Thompson, Seguidores y Berri-Sethi proporcionan contraejemplos *parciales*. Esto es porque, como hemos explicado en el apartado 5.3, las implementaciones basadas en los métodos de Thompson, Seguidores y Berri-Sethi funcionan con autómatas *parciales*, en las que no todas las transiciones tienen que estar presentes. Esto permite detener el proceso en cuanto se descubre que en un autómata existe una transición para un determinado símbolo, pero en el otro autómata no existe tal transición. Aplicando literalmente los métodos explicados en los capítulos 2 y 3, estos métodos podrían ofrecer también siempre contraejemplos totalmente elaborados.

Comparando nuestros resultados con las correcciones del profesor vemos que, efectivamente, los resultados que devolvemos son correctos. Esta herramienta está diseñada para averiguar si dos expresiones regulares son equivalentes o no y, en caso de que no, devuelva un ejemplo de error. No está planteada para devolver todas las discrepancias que existan entre un lenguaje y otro. Por esto es posible observar que las correcciones del profesor son más completas y encuentran más fallos: nuestra aplicación se detiene en cuanto encuentra un motivo de no equivalencia. No obstante, también es destacable que nuestra aplicación encuentra fallos que, a priori, no habían sido apreciados por el profesor.

De esta forma, se estima interesante plantear un nuevo refinamiento de la herramienta que explore todas las posibles discrepancias. Se estima que dicha ampliación no debería suponer una gran dificultad (bastaría agotar todos los objetivos planteados en el algoritmo de Hopcroft-Karp, en lugar de interrumpir el proceso cuando se descubre la primera discrepancia).

6. Conclusiones y trabajo futuro

6.1 – Conclusiones

A continuación, evaluamos el trabajo realizado en relación con los objetivos inicialmente planteados:

1. El primer objetivo planteado fue estudiar los algoritmos necesarios para realizar la comprobación de la equivalencia de especificaciones basadas en expresiones regulares.
   Consideramos que este objetivo ha sido satisfactoriamente abordado.
   Además de estudiar el algoritmo que comprobaba la equivalencia entre dos autómatas finitos deterministas, que era el objetivo, se han estudiado diversos algoritmos que nos sirvieran para construir los autómatas finitos equivalentes a las expresiones regulares (e incluso versiones alternativas) y los algoritmos necesarios para su determinación en caso de que los autómatas resultantes no fueran ya deterministas. Todos ellos están descritos en el capítulo 2 de esta memoria.
2. El segundo objetivo planteado fue diseñar distintos métodos de comprobación de la equivalencia.
   Se ha generado y descrito todo lo necesario para realizar la comprobación de equivalencia mediante cinco secuencias de algoritmos diferentes. En todas las secuencias de algoritmos se han realizado modificaciones para maximizar la "pereza" de las construcciones, que son las versiones que se han implementado en la herramienta. Las modificaciones para añadir "pereza" son las detalladas en el capítulo 3. Por tanto, el grado de consecución de este objetivo es también satisfactorio.
3. El tercer objetivo fue implementar un sistema para la comprobación de la equivalencia de especificaciones basadas en definiciones regulares.
   Se ha creado una herramienta, ajustándonos a los algoritmos y herramientas que se han descrito a lo largo de todo este trabajo, que permite al usuario conseguir lo inicialmente planteado: comparar especificaciones basadas en expresiones regulares. Dichas especificaciones pueden expresarse en un lenguaje razonablemente rico, similar al usado en la asignatura de "Procesadores de Lenguajes" para especificar los aspectos léxicos de los lenguajes de programación. La herramienta proporciona soporte para cargar especificaciones expresadas en dicho lenguaje, así como seleccionar distintos mecanismos de comparación. Como resultado, este objetivo se ha completado también satisfactoriamente.
4. Por último, planteamos realizar una evaluación preliminar del sistema.
   Durante todo el proceso de creación de algoritmos hemos realizado distintas pruebas, con expresiones regulares sencillas, para comprobar su correcto funcionamiento (no se añaden en esta memoria debido a su gran cantidad y a que el mecanismo de descripción de expresiones regulares ha ido cambiando a lo largo del desarrollo).

Sobre la herramienta final, y gracias a la batería de pruebas proporcionadas por nuestro director de TFG, hemos podido comprobar el correcto funcionamiento de nuestra aplicación, tal y como se muestra en el capítulo 5.
Por tanto, este último objetivo también se ha completado satisfactoriamente.

Como resultado, consideramos que hemos completado con éxito todos los objetivos que nos propusimos al comenzar este proyecto.

6.2 – Conclusions

In the following, we evaluate the accomplished work in relation to the initially stated objectives:

1. The first objective was to study the algorithms needed to perform equivalence checking of specifications based on regular expressions.
   We consider that this objective has been successfully accomplished.
   In addition to studying the algorithm that checked the equivalence between two deterministic finite automata, which was the objective, we have studied other algorithms that could be used to construct equivalent finite automata from regular expressions (and even alternative versions) and the algorithms required for their determination in the case that the resulting automata were non-deterministic. All of them are described in chapter 2 of this document.
2. The second objective was to design different equivalence checking methods.
   Everything that is needed to perform equivalence checking has been generated and described using five different sequences of algorithms. In all the sequences of algorithms, we have made modifications to maximize the "laziness" of the constructions, which are the versions that have been implemented in the final tool. These modifications to add "laziness" are detailed in chapter 3. Therefore, the degree of achievement of this objective is also satisfactory.
3. The third objective was to implement a system for checking the equivalence of specifications based on regular definitions.
   This tool has been created following the algorithms and tools that have been described throughout this work, which allows the user to achieve the initial objective: comparing specifications based on regular expressions. These specifications can be expressed in a relatively rich language, which is like the kind of language used to specify the lexical aspects of programming languages in the "Language processors" course at UCM. The tool provides support for loading specifications expressed in such language, as well as to select different comparison mechanisms. Therefore, this objective has also been successfully completed.
4. Finally, we considered performing a preliminary evaluation of the system.
   Throughout the process of creating algorithms, we have done different tests, with simple regular expressions, to check the tool's correct performance (which are not added in this document, due to their large number and to the fact that the mechanism for describing regular expressions has been changed several times throughout the development).

Regarding the final tool, and thanks to the set of tests provided by our degree project supervisor, we have been able to check the correct behavior of our application, as shown in chapter 5.
Therefore, this last objective has also been completed with success.

Finally, we consider that we have successfully completed all the objectives that we set ourselves at the beginning of this project.

6.3 – Trabajo futuro

Pensando en otras funcionalidades que podría tener nuestra aplicación, se nos ocurren las siguientes ideas:

- Proporcionar diagnósticos más completos, en caso de no equivalencia, mediante el agotamiento de todos los objetivos planteados durante el proceso de comprobación de la equivalencia, tal y como se desprende del análisis de resultados realizado en el capítulo 5.
- Mejoras arquitectónicas en el núcleo de la aplicación, utilizando, por ejemplo, el patrón de diseño *estrategia* (ver [12]) para ensamblar las distintas combinaciones de algoritmos.
- Ampliación de la interfaz con las siguientes características:
    - Un método de uso adicional en el que puedan introducirse dos expresiones regulares y comprobar su equivalencia sin necesidad de cargarlas a través de especificaciones descritas en ficheros.
      Esto permitiría a estudiantes de cursos inferiores utilizarla para comparar solo expresiones concretas.
    - Un método de carga adicional por el que se puedan escribir las especificaciones dentro de la propia herramienta y que, cuando se desee, puedan guardarse en un archivo con el mismo formato con el que se cargarían. Esto permitiría hacer comparaciones a medida que se van describiendo las especificaciones sin el tedio de escribirlas en archivos, luego guardarlas y cargarlas cada vez que se haga un mínimo cambio. La opción de guardar permitiría poder seguir fácilmente con el trabajo en otro momento.
- Uso como aplicación Web: consistiría en una aplicación web en la que el docente pueda definir una especificación y el estudiante pueda enviar la suya para que se realice la comparación sin llegar a ver la especificación del docente.
- Entrada: crear un lenguaje para el archivo de entrada más sencillo e intuitivo. Contemplar, asimismo, nuevos tipos de expresiones (por ejemplo, rangos complementarios del tipo [^ …]: símbolos *no* incluidos en el rango).
- Revisar si algunas interacciones del usuario con la aplicación pueden ser confusas y, en caso afirmativo, mejorar la interfaz de usuario.

6.4 – Future work

Thinking about other features that our application could have, the following ideas come to our mind:

- Providing more complete diagnostics, in case of non-equivalence, through the running out of all the objectives raised during the equivalence checking process, as shown in the analysis of results carried out in chapter 5.
- Architectural improvements in the core of the application, using, for example, the *Strategy* design pattern (see [12]) to assemble the different combinations of algorithms.
- User interface extension with the following features:
    - An additional method of use where two regular expressions can be typed in and checked for equivalence without the need to load them through files written with specifications.
      This would allow students in other courses to use it as well to compare only specific expressions.
    - An additional loading method whereby specifications can be written within the tool itself and, whenever desired, these can be saved to a file as the same format as which they would be loaded. This would allow running comparisons alongside describing the specifications without the tedium of writing them to files, then save and reload them each time there is a small change. The save option would allow for an easy way to resume the work at the next time.
- Use as a web application: it would be a web application where the teacher can define a specification and the student can submit theirs for comparison and not be able to see the teacher's specification.
- Input: create a language for the input file that is simpler and more intuitive. Also contemplate new types of expressions (e.g., complementary ranges like [^…]: symbols that are not included in the range).
- Examine whether some user interactions with the application can be confusing and, if so, improve the user interface.

7. Contribución personal

7.1 – Raúl Benito Montoro

A principios de curso, se hizo una reunión con el director del proyecto donde nos explicó de qué trataría el TFG y el primer paso a seguir: estudiar y describir los algoritmos de creación y determinación de los autómatas y el algoritmo de comprobación de equivalencia. Yo me encargué de describir los siguientes algoritmos:

- Algoritmo de derivadas para convertir expresiones regulares en AFDs.
- Algoritmo de las derivadas parciales, que mejora el algoritmo de las derivadas para evitar los problemas de normalización de expresiones, aun a expensas de producir AFNs en lugar de AFDs.
- Algoritmo de Berry-Sethi, que refina también el algoritmo de las derivadas para evitar los problemas relativos a la normalización (aunque produce también AFNs, en lugar de AFDs).

Después de esto, se hizo otra reunión a la que siguió el comienzo de la implementación de una de las secuencias de algoritmos al completo, en concreto, la secuencia basada en la construcción de Thompson, la construcción de subconjuntos y el algoritmo de Hopcroft-Karp. En esta parte, yo me ocupé de diseñar e implementar el algoritmo que combinaba la determinación de un autómata finito no determinista con λ-transiciones con el algoritmo de Hopcroft-Karp.

Tras otra reunión, dedicamos los siguientes meses a la implementación y prueba de todas las demás secuencias de algoritmos. De los algoritmos importantes descritos en esta memoria, yo me encargué de:

- El algoritmo basado en el método de las derivadas, en su versión ya fusionada con el algoritmo de Hopcroft-Karp.
- El algoritmo basado en el método de las derivadas parciales, ya en su versión fusionada con la determinación de AFNs sin λ-transiciones y el algoritmo de Hopcroft-Karp.
- Los algoritmos de lambda-cierre y determinación de AFNs con λ-transiciones.

En enero, con la llegada de los exámenes, el proyecto se dejó un poco aparcado, aunque ya disponíamos de un programa simple (sin interfaz siquiera) al que podíamos introducirle expresiones regulares sueltas y nos devolvía el resultado de la comprobación de equivalencia a través de las cinco secuencias de algoritmos programadas.

Tras los exámenes, reanudamos el proyecto. En esta fase, me encargué de:

- Diseñar e implementar la interfaz de usuario al completo, tanto la ventana principal como la de ayuda y el envío al controlador de las expresiones regulares marcadas y la selección de algoritmos y métodos que hiciera el usuario.

- Escribir el analizador léxico para el lenguaje de especificación y una primera versión del sintáctico (sin el constructor de árboles de sintaxis abstracta) que se utilizarían en la introducción de especificaciones basadas en expresiones regulares.

Nada más acabar, comenzamos a escribir la memoria. Para ello:

- Cada miembro del grupo empezó a escribir los apartados de los que se había encargado en la implementación (algoritmos y secuencias de algoritmos).
- Yo me encargué de preparar todos los apartados comunes de la memoria: todos aquellos que no trataran de algoritmos o secuencias a excepción del apartado de arquitectura del sistema y la traducción de la introducción y las conclusiones.
- Por último, también me he encargado de llevar a cabo varias revisiones de la memoria, y de colaborar con el director del trabajo en el refinamiento de los aspectos formales del mismo. El director ha propuesto muchos cambios relevantes en relación con dichos aspectos, especialmente en los capítulos 2 y 3, y yo me he encargado de su validación e incorporación a la versión final de la memoria.

Simultáneamente a la escritura de la memoria, se han ido arreglando algunos fallos que encontramos en la aplicación y se hicieron sugerencias de mejora que se han ido implementando a lo largo de los últimos dos meses.

A continuación, realizo un resumen de lo más importante de mi contribución a este trabajo:

- Algoritmos principales implementados en la herramienta:
    - Algoritmo de Hopcroft-Karp en su versión conjunta con determinación por subconjuntos (para autómatas con λ-transiciones).
    - Algoritmo de derivadas en su versión conjunta con el algoritmo de Hopcroft-Karp.
    - Algoritmo de derivadas parciales en su versión conjunta con determinación y el algoritmo de Hopcroft-Karp.
    - Algoritmo de determinación de un estado con λ -transiciones
    - Algoritmo de lambda-cierre
    - Interfaz completa de usuario.
    - Analizador léxico y versión inicial del sintáctico.
- Puntos de la memoria escritos:
    - Capítulo 1 en español.
    - Capítulo 2: aspectos preliminares, algoritmos de derivadas, derivadas parciales y Hopcroft-Karp.
    - Capítulo 3.
    - Capítulo 4 excepto 4.1: arquitectura del sistema.
    - Capítulo 5.
    - Capítulo 6 en español.
    - Capítulo 7: mi participación.

7.2 – Xukai Chen

En la primera reunión con el director se establecieron los algoritmos que necesitaría el programa para llevar a cabo el desarrollo del proyecto. Una vez concluida, nos pusimos a investigar sobre ellos. Aunque el profesor nos ofreció información de algunos, se han encontrado también otros algoritmos que presentan mejores eficiencias. Tras varias reuniones resolviendo las dudas que nos surgieron, establecimos un plan de trabajo.

A partir de entonces, a mediados de octubre, Raúl y yo nos hemos repartido las tareas para escribir los pseudocódigos de los diferentes algoritmos. A este respecto, yo me he encargado:

- Del algoritmo de Hopcroft-Karp, que permite comprobar la equivalencia de dos AFDs simulando simultáneamente sus posibles ejecuciones.
- De la determinación de AFNs con y sin λ-transiciones.
- Del algoritmo de Thompson y del algoritmo de seguidores, el primero para traducir expresiones regulares en λ-AFNs, y el segundo un refinamiento del algoritmo de Thompson que produce AFNs sin λ-transiciones.

Una vez terminados estos algoritmos, hemos quedado para compartir informaciones y entender cada uno la parte correspondiente del otro. Aunque este reparto no se ha llevado hasta las implementaciones, nos ha servido para tener ambos las ideas fundamentales de los algoritmos.

Para tener una primera versión del programa que pueda realizar la mínima función del proyecto, se ha decidido escoger solo un algoritmo de transformación de expresión regular en AFN, y olvidarnos en su momento del *parser*. Tras hablar entre nosotros, hemos creído que nos conviene dividir el programa en varios módulos e implementar de forma modular. Yo me he encargado de las siguientes partes:

- Diseño de la estructura de los árboles de sintaxis abstracta que representan las expresiones regulares.
- Implementación de un *parser* sencillo basado en pila, que nos permitió realizar las validaciones y evaluaciones iniciales de los algoritmos implementados.
- Desarrollo del algoritmo de Thompson al que he hecho alusión anteriormente, y que es el algoritmo más sencillo y típico para transformar expresiones regulares en AFNs.

En diciembre, ya teníamos implementado un programa sencillo que devolvía resultados. Tras enseñárselo al director, se establecieron los siguientes objetivos:

- Continuar añadiendo diferentes algoritmos de transformación de expresiones regulares en AFNs.
- Dar soporte a expresiones regulares más complejas (opcionalidad, rangos de símbolos y Kleen positivo).

Seguimos con la misma idea de repartir los trabajos planificados. Para ello, yo me he ocupado de:

- Diseñar los algoritmos de seguidores y Berry-Sethi.

- Dar soporte a las expresiones regulares más complejas.

Debido al periodo de exámenes del primer cuatrimestre, el proyecto se vio retrasado. La siguiente reunión la tuvimos a finales de febrero, cuando ya habíamos abordado todos los objetivos. En esta reunión, el director nos recomendó:

- Ir redactando la memoria o parte de ella.
- Añadir una interfaz gráfica para que el programa fuera más usable, soportando los distintos métodos de comparación y selección de definiciones a comparar, y un *parser* basado en los conocimientos de la asignatura "Procesadores de Lenguajes".

Esta vez, me encargué de desarrollar el *parser* utilizando las mismas herramientas que se emplearon en las prácticas de la asignatura de "Procesadores de Lenguajes" y del diseño y la implementación de la estructura del modelo vista-controlador.

Durante la implementación del *parser* tuvimos problemas a la hora de diseñar el lenguaje de especificación. Por ejemplo, no sabíamos cómo diferenciar entre las variables y las expresiones regulares. Consultamos a nuestro director para solventar estos problemas. Después, se estableció el lenguaje de especificación tal como está ahora, y me encargué de su implementación final.

A mediados de abril, ya teníamos un programa con la apariencia actual y ejecutable. Hemos seguido arreglando los *bugs* que surgían y preparando contenidos de la memoria. Los contenidos dependen de lo que ha hecho cada uno. A mí me ha correspondido la redacción de las partes relacionadas con aquellos módulos que he estado implementando.

A mediados de mayo, y con el programa ya terminado, le he entregado mis contenidos de la memoria a Raúl, que se ha encargado de la integración, para que los junte en la memoria final.

Una vez obtenida la primera versión de memoria, se la hemos entregado al director para la revisión, y después nos hemos reunido con él para ver los fallos y sugerencias para mejorar la memoria. Ambos nos hemos puesto seguidamente a arreglar la memoria. Aparte de esto, yo me he encargado de traducir al inglés los capítulos de Introducción y Conclusiones

Simultáneamente hemos decidido que nuestro programa se alojará en Github de forma pública bajo una licencia MIT. Yo he liderado este esfuerzo. Para ello:

- Me he ocupado de redactar un *readme* para el repositorio.
- He añadido los créditos correspondientes.
- He confeccionado una pequeña guía de uso e instalación, descripción del repositorio, y los diferentes ejemplos de prueba.
- Por último, he preparado también un ejecutable Java (.jar) para facilitar la ejecución de nuestra herramienta.

8. Bibliografía

[1] K. Thompson. “Regular expression search algorithm”. Communications of the ACM 11 (6) 419–422. 1968.

[2] S. Sippu, E. Soisalon-Soininen, Parsing Theory: I Languages and Parsing, EATCS Monographs on Theoretical Computer Science, vol. 15, Springer-Verlag, New York, 1988.

[3] L. Ilie, S. Yu. “Follow Automata”. Information and Computation 186, 140–162. 2003

[4] Janus A. Brzozoski. “Derivatives of Regular Expressions”. Journal of the ACM 11(4), 481–494. 1964.

[5] V. Antimirov. “Partial Derivatives of Regular Expressions and Finite Automaton Constructions”. Theoretical Computer Science, 155(2), 291-319. 1996.

[6] J.M. Champarnaud y D. Ziadi. “Canonical Derivatives, Partial Derivatives and Finite Automaton Constructions”. Theoretical Computer Science 289(1), 137-163. 2002.

[7] G. Berry, R. Sethi. “From regular expressions to deterministic automata”. Theoretical Computer Science 48, 117–126. 1986

[8] Watson, B. W. A taxonomy of finite automata construction algorithms. (Computing science notes; Vol. 9343). Technische Universiteit Eindhoven. 1993

[9] Briiggemann-Klein, A. “Regular expressions into finite automata”. Theoretical Compute Science 120, 197-213. 1993

[10] A.V. Aho, M.S. Lam, R. Sethi, J.D Ulman. “Compilers: Principles, Techniques and Tools”. 2nd Edition. Addison-Wesley. 2006.

[11] Norton, Daphne, "Algorithms for testing equivalence of finite automata, with a grading tool for JFLAP". PhD. Thesis. Rochester Institute of Technology. 2009.

[12] John E. Hopcroft y R. M. Karp. "A Linear Algorithm for Testing Equivalence of Finite Automata". Tech. Report. Cornell University. 1971.

[13] E. Gamma, R. Helm, R, Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1994.

[14] D. E. Knuth. Semantics of Context-free languages: Correction. Math. Systems Theory, 5(1): 95-96, 1971.

[15] B. Temprado. Un Enfoque Metalingüístico al Procesamiento de Documentos XML. Tesis Doctoral. UCM. 2015