Internal Report 2012–07 July 2012

Universiteit Leiden
Opleiding Informatica

Efficient Equivalence Checking

of

Regular Expressions

Leroy van Delft

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Leiden University

Efficient Equivalence Checking of Regular Expressions

Author:
Leroy van Delft

Supervisors:
Dr. M. Bonsangue
J. Rot MSc

July 10, 2012

Contents

1 Introduction

Regular expressions have many uses in computer science, including: protocol checking and pattern matching, moreover regular expressions are used in compilers. Furthermore regular expressions can be used as an abstract programming language. The use of regular expressions in these applications most of the time comes down to answering questions such as: Are two different syntactic descriptions of a language identical? But also other questions which seem trivial, but are not, are commonly solved via regular expressions. One such question would be: Is a language empty? Most of these questions involve solving equivalence for two regular expressions.

In this paper we present an implementation which can check equivalence in an efficient way. The implemtation is based on the creation of a product automaton. This is created with the use of derivatives of regular expressions. Besides the implemented product automaton the goal was to make it more efficient. To gain this efficiency we use the properties of the equivalence operator. In this thesis first of all basic definitions for regular expressions are given. Furthermore definitions for derivatives are given, followed by an explaination of techniques for the creation of finite automata. This is all found in Section 2. In Section 3 an introduction to the programming language Maude will be given. Maude is the programming language in which the implementation is made. The implementation is discussed in Section 4, it shows the way the implementation was established and discusses tradeoffs. A comparison between multiple variants of the implementation has been done, in Section 5 these are presented and explained. In Section 6 the results are discussed and conclusions are drawn from these results. The code for the implementation can be found in the Appendix.

2 Theory and Techniques

The definitions introduced in this section are from [1, 2].

An alphabet Σ is a finite set of symbols. Throughout this paper the alphabet used will be a, b unless stated otherwise. The set of all strings over Σ is defined as Σ^* and we use u, v to represent strings. A language L of Σ is a set of strings, $L \subseteq \Sigma^*$. For a string u, $|u|$ stands for the length (the number of symbols) of u. The null-sting ε is a string over Σ and by definition $|\varepsilon| = 0$.

2.1 Regular Expressions and Regular Languages

The syntax of regular expressions (RE) includes the standard operations: concatenation, Kleene-closure and alternation (also called logical-or). In addition it is extended with the Boolean operation *and*. Throughout this paper we will use r, s and t to represent regular expressions.

Definition 2.1. A Regular Expression is defined as:

i) $\emptyset$ is a regular expression

ii) ε is a regular expression

iii) a is a regular expression, for every alphabet symbol $a \in \Sigma$

iv) $r \cdot s$ is a regular expression **if** r and s are regular expressions

v) $r + s$ is a regular expression **if** r and s are regular expressions

vi) $r\&s$ is a regular expression **if** r and s are regular expressions

vii) r^* is a regular expression **if** r is a regular expression

viii) Nothing else is a regular expression

Definition 2.2. The *language* of a regular expression r is a set of strings $L(r) \subseteq \Sigma^*$ generated by the following rules:

$L(\emptyset) = \emptyset$
$L(\varepsilon) = \{\varepsilon\}$
$L(a) = \{a\}$, $a \in \Sigma$
$L(r \cdot s) = \{u \cdot v \mid u \in L(r) \text{ and } v \in L(s)\}$
$L(r + s) = L(r) \cup L(s)$
$L(r\&s) = L(r) \cap L(s)$
$L(r^*) = \{\varepsilon\} \cup L(r \cdot r^*)$

All languages that can be described by a formula involving the operations of union, concatenation and Kleene-closure are called *regular languages*. In other words regular languages are those languages that can be described by regular expressions according to Definition 2.2.

2.2 Finite Automata

A finite automaton is a model of a computation device, which acts as a language acceptor. In this paper only the deterministic finite automaton is needed, which is defined as follows:
Definition 2.3 A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where
Q is a finite set of *states*
Σ is a finite *input alphabet*
$q_0 \in Q$ is the *initial* state
$A \subseteq Q$ is the set of *accepting states*
$\delta : Q \times \Sigma \to Q$ is the *transition function*

We define the extended transition function $\delta^* : Q \times \Sigma^* \to Q$ as follows:
$\delta^*(q, \varepsilon) = q$
$\delta^*(q, au) = \delta^*(\delta(q, a), u)$

The language of a DFA M is $L(M) = \{u \in \Sigma^* \mid \delta^*(q_0, u) \in A\}$.

Having these defintions we can state Kleene's Theorem: For every alphabet Σ, every regular language over Σ can be accepted by a finite automaton. According to Kleene's Theorem the reverse is also true: For every finite automaton $M = (Q, \Sigma, q_0, A, \delta)$, the language $L(M)$ is regular. Proofs for both parts of this theorem can be found in [1].

2.3 Regular Expression Derivatives

Before we can define the derivative rules for regular expressions we need the help of a function. This function we call ν, which maps REs to REs. We say a regular expression r is *nullable* if the language that it defines contains the empty string, which means that $\varepsilon \in L(r)$.
Definition 2.4 The ν function can be calculated using the following rules:
$\nu(\varepsilon) = \varepsilon$
$\nu(a) = \emptyset$
$\nu(\emptyset) = \emptyset$
$\nu(r \cdot s) = \nu(r) \& \nu(s)$
$\nu(r + s) = \nu(r) + \nu(s)$
$\nu(r \& s) = \nu(r) \& \nu(s)$
$\nu(r^*) = \varepsilon$

The function ν has the property that: $\nu(r) = \begin{cases} \varepsilon & \text{if } r \text{ is nullable} \\ \emptyset & \text{otherwise} \end{cases}$
The notion of a language derivative, with respect to a symbol $a \in \Sigma$ corresponds to the language that includes only those strings with a leading symbol a in L.
Definition 2.5 The derivative of a language $L \subseteq \Sigma^*$ with respect to a symbol $a \in \Sigma$ is defined to be $\partial_a L = \{u \mid a \cdot u \in L\}$.
A set of rules, due to Brzozowski [3], is used to compute the derivative of a regular expression. The following rules compute the derivative of a regular expression with respect to a symbol $a \in \Sigma$:
$\partial_a(\varepsilon) = \emptyset$
$\partial_a(a) = \varepsilon$

$\partial_a(b) = \emptyset$, for $b \neq a$
$\partial_a(\emptyset) = \emptyset$
$\partial_a(r \cdot s) = \partial_a(r) \cdot s + \nu(r) \cdot \partial_a(s)$
$\partial_a(r + s) = \partial_a(r) + \partial_a(s)$
$\partial_a(r \& s) = \partial_a(r) \& \partial_a(s)$
$\partial_a(r^*) = \partial_a(r) \cdot r^*$

The derivative of a regular expression with respect to a symbol a has the following property: $L(\partial_a(r)) = \partial_a(L(r))$

We can extend these derivatives to strings, as follows:
$\partial_\varepsilon(r) = r$
$\partial_{au}(r) = \partial_u(\partial_a(r))$

These derivatives can be used to check if a string is contained in the language of a regular expression.
For every string u and every regular expression r: $u \in L(r) \Leftrightarrow \varepsilon \in L(\partial_u(r))$. This is exactly true when $\varepsilon = \nu(\partial_u(r))$.

To check if a string is contained in the language of a regular expression we use a new relation $\sim$. If $r \sim u$ then we say r matches 'ok'. The definition of this relation is as follows.
Definition 2.6
$r \sim \varepsilon \Leftrightarrow \nu(r) = \varepsilon$
$r \sim au \Leftrightarrow \partial_a(r) \sim u$
The function ν is required because it is needed to check if $\varepsilon \in L(\partial_u(r))$. As stated before this is exactly true when $\varepsilon = \nu(\partial_u(r))$.

Example 2.1 $abb \in L(a \cdot b^*)$

$$\begin{aligned} a \cdot b^* \sim abb &\Leftrightarrow \partial_a(a \cdot b^*) \sim bb \\ &\Leftrightarrow b^* \sim bb \\ &\Leftrightarrow \partial_b(b^*) \sim b \\ &\Leftrightarrow b^* \sim b \\ &\Leftrightarrow \partial_b(b^*) \sim \varepsilon \\ &\Leftrightarrow b^* \sim \varepsilon \\ &\Leftrightarrow \nu(b^*) = \varepsilon \end{aligned}$$

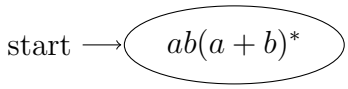
If the RE does not match the string u, we reach the derivative that is $\emptyset$, and stop.

These derivatives can be further extended, to derivatives of regular expressions with respect to another regular expression.
Definition 2.7
$\partial_\emptyset(r) = \emptyset$
$\partial_{s+t}(r) = \partial_s(r) + \partial_t(r)$
$\partial_{s \cdot t} = \partial_t(\partial_s(r))$
$\partial_{s^*}(r) = r + \partial_{s^*}(\partial_s(r))$ **if** $r \neq \partial s(r)$
$\partial_{s^*}(r) = r$ **if** $r = \partial_s(r)$

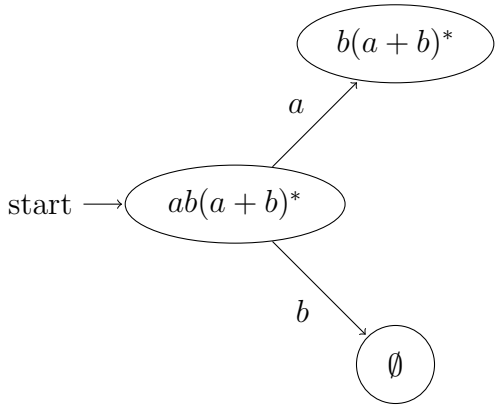
This results in: $L(s) \subseteq L(r) \Leftrightarrow \varepsilon \in L(\partial_s(r))$.
There are two rules for Kleene-closure to ensure termination. If only the first rule is defined calculation of the derivative might result in endless recursion. It will do that as $\partial_{s^*}(r)$ will eventually be minimal eg. will not change over s. This results in the fact that endless repetition of the same result will occur. It will result in $t + r + r + \cdots$, where t is everything that has been correctly derivited untill that point. We know we are minimal if $r = \partial_s(r)$ yields true, this is the reason for the second rule as it will ensure termination.

2.4 DFA construction

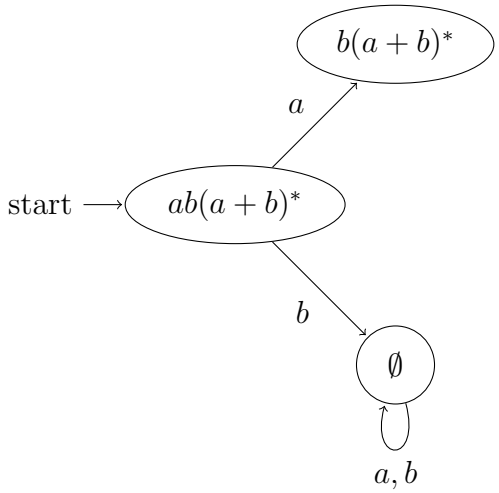
Another thing that can be done using the Brzozowski derivatives is the creation of a deterministic finite automaton (DFA) using these derivatives. The idea behind this is that the derivative with resprect to a symbol a of a regular expression corresponds with 'reading' that symbol in the deterministic automaton. This will be used to create transitions and states in the DFA. The initial state will be the regular expression we want to make a DFA for, the other states are the derivatives with respect to all symbols $a \in \Sigma$. Transitions with a symbol a are then added between two states (say q_1 and q_2) if $\partial_a(q_1) = q_2$.
Example 2.2 Creation of a DFA for $ab(a + b)^*$ (all words starting with ab).
First of all we can add the state for $ab(a + b)^*$.



Next we calculate the derivatives for the first state:
$\partial_a(ab(a + b)^*) = b(a + b)^*$
$\partial_b(ab(a + b)^*) = \emptyset$
Two new states and two new transitions can be added now.



The next extension will be the simple one, the derivatives for $\emptyset$.
$\partial_a(\emptyset) = \emptyset$
$\partial_b(\emptyset) = \emptyset$

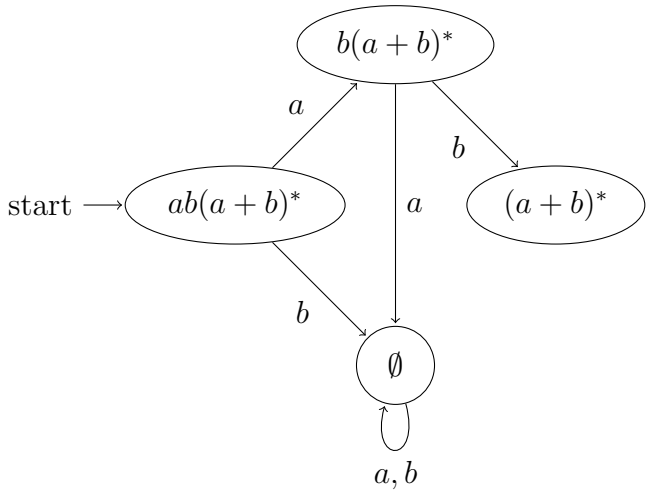


Note that no new state for $\emptyset$ is created. A new state may only be added to the automaton if it is not already present in the automaton. If this restriction would not be there this process would not terminate.
The only state left now to extend is $b(a+b)^*$.

$\partial_a(b(a+b)^*) = \emptyset$

$\partial_b(b(a+b)^*) = (a+b)^*$

This yields the following state and transitions.

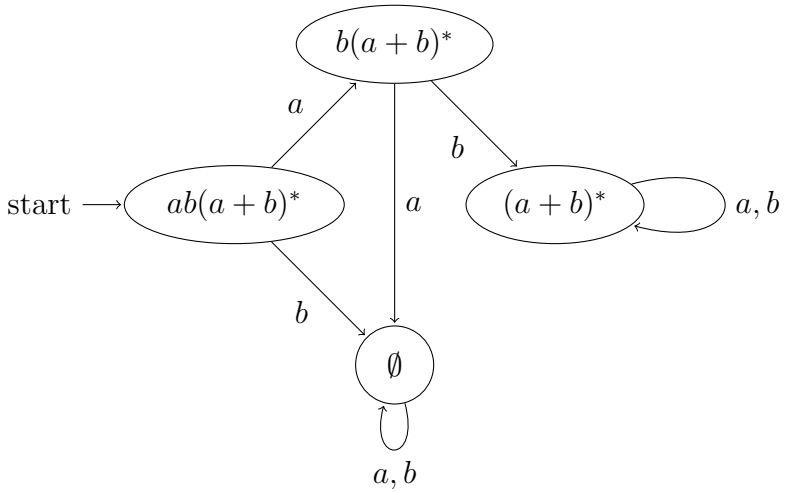


For the newly added state transitions still need to be added and maybe also new states.

$\partial_a((a+b)^*) = (a+b)^*$

$\partial_b((a+b)^*) = (a+b)^*$

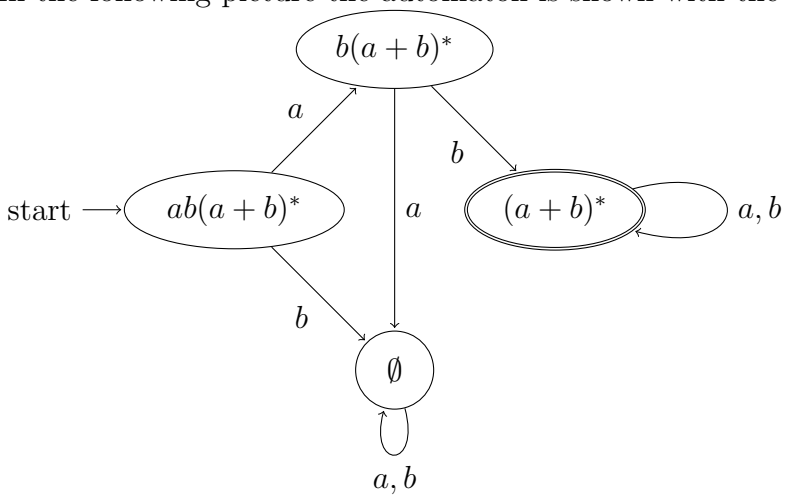
This finishes the automaton as follows.



While the automaton is now complete a detail is missing, which is the fact to determine which of the states in the new DFA are accepeting. The can be done by checking if the regular expression of a state is nullable. If $\nu(r) = \varepsilon$ then the state is accepting, if $\nu(r) = \emptyset$ then the state is non-accepting.

$\nu(ab(a+b)^*) = \emptyset$
$\nu(b(a+b)^*) = \emptyset$
$\nu((a+b)^*) = \varepsilon$
$\nu(\emptyset) = \emptyset$

In the following picture the automaton is shown with the accepting states.



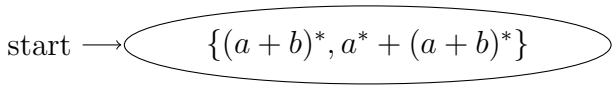
We now have an automaton that accepts the same language as the regular expressions $ab(a+b)^*$. As shown above this method works, however one point of interest occurs. That is the question whether or not this method will teminate. Brzozowski provided a proof in his paper [3], that regular expressions can only have a finite number of derivatives modulo associativity, associativity and idempotence (ACI) of the + operator. This also guarantees that the construction of the deterministic finite automaton will terminate.

2.5 Product Automata

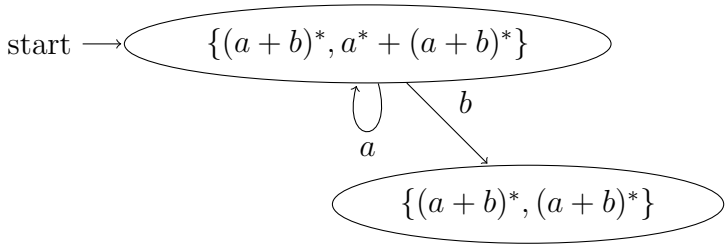
The construction explained in the previous section leads something more interesting which is equivalence checking of two regular expressions. Before this can be done however a new

definition is required.
Definition 2.8 Two regular expressions, r and s, are equivalent, denoted as $r \equiv s$, if $L(r) = L(s)$.

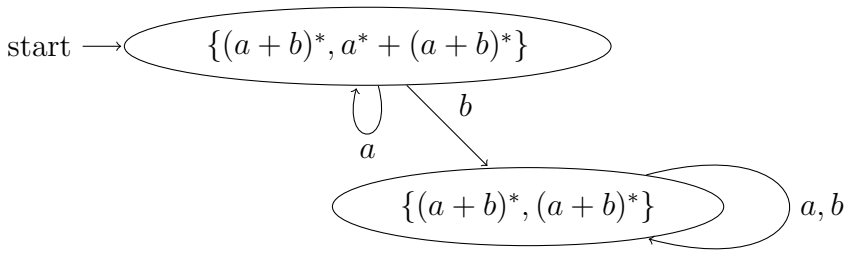
Equivalence checking will be done with a *product automaton*, which is also know as *bisimulation*. A product automaton (PA) is similar to a DFA, however the way it is used is that every state contains a pair of regular expressions. The construction of such a PA is therefore based on the construction of a DFA expained in the previous section. The initial state contains the pair of the original regular epressions, which need to be checked for equivalence. New states are added by calculating the derivatives of both REs in a state. A state is can be added if both regular expressions are nullable or both are not (both either ε or $\emptyset$). This condition also needs to hold for the initial state. The condition is required to make sure they accept the same words. If one of the regular expressions accepts a word the other regular expression does not, then they are not equivalent and no valid PA can be produced. Transitions are added in the same way as defined in the DFA construction.
Example 2.3 Product Automaton for $\{(a+b)^*, a^* + (a+b)^*\}$
The first thing is to check if $\nu((a+b)^*) = \nu((a^* + (a+b)^*))$. Indeed $\nu((a+b)^*) = \varepsilon$ and $\nu(a^* + (a+b)^*) = \varepsilon$. For this reason we can add the state.



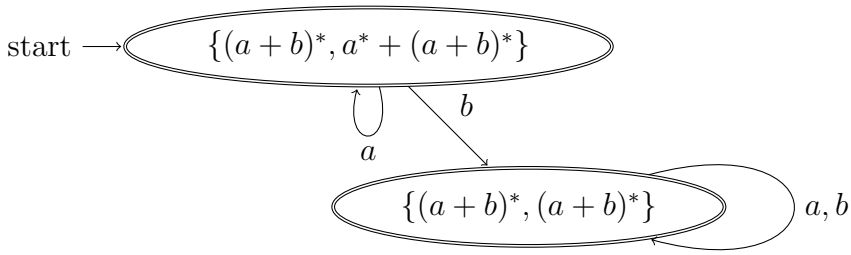
Next up is computing the derivatives of both REs.
$\partial_a((a+b)^*) = (a+b)^*$
$\partial_a(a^* + (a+b)^*) = a^* + (a+b)^*$
$\partial_b((a+b)^*) = (a+b)^*$
$\partial_b(a^* + (a+b)^*) = (a+b)^*$
This means we get the following pairs: $\{(a+b)^*, a^* + (a+b)^*\}$ and $\{(a+b)^*, (a+b)^*\}$, the first one is the original pair and does not need to be added. Of course the new pair needs to be checked: $\nu((a+b)^*) = \nu((a+b)^*) = \varepsilon$.



The pair $\{(a+b)^*, (a+b)^*\}$ still needs to be extended. This is trivial as the derivatives are already known. So this yields the following result.



The only thing to determine next is which states are accepting. A state is accepting if both its expressions are nullable, and the state is non-accepting when they're not. We already know that all four regular expressions in the two states are nullable.



Because we know that regular expressions can only have a finite number of derivatives. This guarantees that the construction of the product automaton will terminate.

As shown a product automaton can be used to check whether $r \equiv s$. The construction can be altered slightly to check whether $L(r) \subseteq L(s)$, for regular expressions r and s. The algorithm for creating this new automaton is identical to the creation of the product automaton, except for one change. Because the subset operation is different from equivalence it reults in that a different check is required when adding states. In the creation of the product automaton it is checked, everytime a new state is to be added, whether $\nu(r) = \nu(s)$. When checking for $L(r) \subseteq L(s)$ it is not required that for every new state $\nu(r) = \nu(s)$ holds. It is now required that for every new state it holds that $\nu(r) = \nu(s)$ or $\nu(s) = \varepsilon$. This means that when a string $u \in L(r)$ then it must hold that $u \in L(s)$, however when a string $t \in L(s)$ then it is not required that $t \in L(r)$.

2.6 Efficient Equivalence Checking

While we now have a method for equivalence checking of regular expressions, this method is not as efficient as it could be. The goal is to be more efficient in space, by minimizing the number of states. The way to achieve this goal is to make use of the following properties of equivalence relation:

- Reflexivity, $r \equiv r$
- Symmetry, if $r \equiv s$, then also $s \equiv r$
- Transitivity, if $r \equiv s$ and $s \equiv t$, then also $r \equiv t$

Besides these three 'basic' properties another property is added. This property is called the +-closure which is defined as follows:

- If $r \equiv p$ and $t \equiv s$, then also $r + t \equiv p + s$

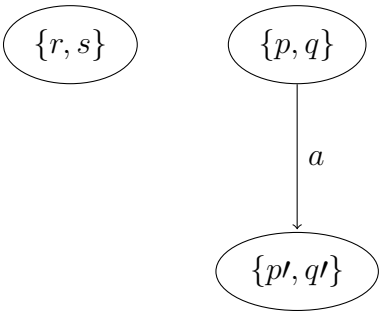
The least equivalence relation including a relation R and $+$-closure is called the closure of R

When producing a product automaton it is checked whether a state $\{r, s\}$ is already present in the product automaton to avoid an infinite loop. With the improved method instead it is checked whether $\{r, s\}$ is in the closure of the current product automaton. We will illustrate this with an example.

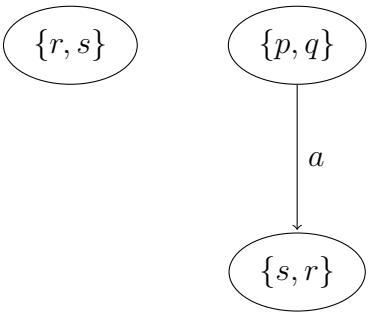
Example 2.4 Symmetry

Let there be a product automaton as follows:

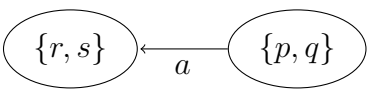
Transitions are left out, for this example it is assumed that there are no states other than the states show in the picture. Let us go and expand the state with from the $\{p, q\}$; this means calculating the derivatives and adding the transitions. Let $\partial_a(p) = p\prime$ and $\partial_a(q) = q\prime$, for a symbol $a \in \Sigma$. This yields:



Now suppose that $\{p\prime, q\prime\} = \{s, r\}$. This yields the following automaton:



This state would be added in the product automaton, as there is no state $\{s, r\}$ yet. In the improved method however we need to check whether $\{s, r\}$ is in the closure. Indeed there is a state $\{r, s\}$, because of symmetry the state $\{s, r\}$ is in the closure. This results in the fact that the new state $\{p\prime, r\prime\}$ does not need to be added. If one wants to have something that resembles an automaton one can now add a new transition as follows:



This resulting automaton is no longer considered a product automaton.

3 Maude

For the implementation I used a programing language called Maude [4]. "So what is Maude? Maude is a programming language that models systems and the actions within those systems." [5]. In this chapter some of the basics of Maude will be explained so the implementation made for this paper can be understood. This section is not intended as a manual or tutorial on how to use Maude. Maude is closest related to logical programming languages such as Prolog. Similar to logical programming languages Maude is based on pattern matching and replacing. Unlike high-end languages such as C++ and Java, Maude programs are not compiled but interpreted. In the Maude environment we can load programs (called modules) and then ask questions regarding what is defined in the module.

Example 3.1 A simple Maude module (taken from [5])

```
fmod Animals is
  sorts Animal Dogs .
  subsort Dog < Animal .
  ops bloodhound terrier pitbull schnauzer : -> Dog .
  op penguin : -> Animal .

  op breed : Dog Dog : -> Dog .
endfm
```

In this module we can breed dogs with each other. In the Maude-environment the following expression could be typed: `reduce breed(bloodhound, pitbull)`; this will result in a positive answer and return you `Dog`. Another expression, `reduce breed(terrier, penguin)` will not give a positive answer as `penguin`, because the sort `Animals` is not in a subsort of `Dogs`. As `breed` is defined only with two dogs as parameters, the query will fail. In Maude the command `reduce` (or `red`) is used to reduce an expression as far as possible. Once no more reductions are possible Maude terminates and the result is given.

3.1 Equations & Rewrite Laws

One of the most important features of Maude is the ability to use *equations*. The purpose of an equation is to provide Maude with rules to simplify an expression. The best way to see how these equations work, is via the use of an example.

Example 3.2 Equations

```
fmod Operations is
  protecting NAT

  op _+_ : Nat Nat -> Nat [comm].

  var N : Nat .
  eq 0 + N = N .
endfm
```

In this example a module which defines the natural numbers (called `NAT`) has been imported. Under the assumption that `NAT` allows for the use of natural numbers, but contains

no operators we can define the addition operator. Because of the added equation (keyword `eq`) we can now reduce the following expression: `red 0 + 5`, which will return 5. Note here that the expression: `red 5 + 0` will also be reduced to 5. This is 'correct' because we have specified that addition is commutative via `[comm]`.

Equations are very useful in Maude as they create and map out structures, however the real power of Maude is about transitions that occur within and between structures. These transitions are mapped in *rewrite laws*. As Maude is closely related to a logical programming language, the main strength of Maude lies in rewriting logic, hence rewrite laws. Rewrite laws are used to describe changes between different *states* with the use of *transitions*.

Example 3.3 Rewrite Laws (taken from [5])

```
fmod Smoking is
  sort State .

  op cigarette : -> State [ctor] .
  op butt : -> State [ctor] .
  op __ : State State -> State [ctor assoc comm] .
  rl [smoke] : c => b .
  rl [makenew] : b b b b => c .
endfm
```

This example is based on a puzzle. If there is a person who smokes, but who also can make a new cigarette from 4 butts (what is left over after smoking a cigarette), how many cigarettes can he smoke if he starts with x cigarettes? This module will not solve the puzzle but can simulate it. Each rewrite law is indicated with `rl`, the part between `[]` is treated as comments. So we can call the Maude environment: `rewrite c c c c`, which simulates smoking 4 cigarettes. In the Maude environment `rewrite` (or `rew`) is used instead for rewrite laws instead of `reduce`. The result is `b`. This is because the 4 initial cigarettes are smoked first, then the 4 butts are rewritten to a cigarette, and that is then smoked, leaving a single butt.

3.2 Motivation for using Maude

The main motivation for using Maude is that it makes the implementation much easier. If the implementation would be done in a more 'conventional' language such as Java or C++ new data structures would need to be defined: for regular expressions and for states and transitions (in the automata). Furthermore many lines of code are necessary for the implementation that can be done in a single line in Maude. Here is an example that makes clear how efficient the implementation can be done.

Example 3.4 Nullable

Recall the rules for the function ν:

$\nu(\varepsilon) = \varepsilon$
$\nu(a) = \emptyset$
$\nu(\emptyset) = \emptyset$
$\nu(r \cdot s) = \nu(r) \& \nu(s)$
$\nu(r + s) = \nu(r) + \nu(s)$

$\nu(r^*) = \varepsilon$

In Java for example checking if an RE is nullable would require a loop with various if-statements. In Maude the code will look as follows:

```
eq v(1) = 1 .
eq v(0) = 0 .
eq v(A) = 0 .
eq v(E1 - E2) = v(E1) & v(E2) .
eq v(E1 + E2) = v(E1) + v(E2) .
eq v(E *) = 1 .
```

As can be seen the rules are described in exactly the same amount of lines (6) as the mathematical notation. Of course an operations of regular expressions needs to be defined as well as the variables `E1`, `E2` and `E`. This example shows how efficient mathematical functions can be defined in Maude.

While such functions can be implemented easily, one of the more difficult things to implement in Maude is loops. In Maude both while-loops and for-loops are not present, there is something called a `LOOP-MODE`, but this does not reflect loops in other programming languages. The best way to loop in Maude is using recursion. This is already shown in example 3.4, the rules for both concatenation and alternation. For example, in the equation for alternation we can see that `v(r + s)` is reduced to `v(r) + v(s)`. Because Maude uses pattern matching, in the next step of the evaluation `v(r)` will be reduced. The main problem with recursion (in any prgramming language) is to ensure termination. In Maude this can be ensured by adding a condition to equations, which can only be executed if this condition is met. As stated before, Maude will continue rewriting and/or reducing until it is out of options. So endless recursion can be stopped by ensuring that a function is rewritten or reduced to a terminal (such as `eq v(A) = 0 .`).

4 Implementation

For the implementation the goal is to get a working set of modules which can check if two regular expressions describe the same language. Checking equivalence is done via the method of a product automaton described in Section 2. Also the more efficient method (as explained in Section 2.6) is implemented. Before getting to that stage several other aspects related to this need to be implemented. In the final product the following modules are present:

- *RegularExpression*: This module contains all basic operations and definitions related to regulare expressions.
- *Nullable*: The function ν is implemented in this module.
- *Derivative*: Brzozowski derivatives and all the extension thereoff are present in this module.
- *Automaton*: Contains all operation for the construction of a DFA.
- *ProductAutomaton*: Creates a product automaton for two given regular expressions (if possible), to check if two regular expressions are equivalent.
- *Subset*: Checks if the languages of two provided regular expressions (r, s) are a subset $(L(r) \subseteq L(s))$.
- *Equivalence*: Also checks equivalence of regular expressions. Uses transitivity, reflexivity, symmetry and +–closure to be more efficient.

The module RegularExpression will not be discussed as this is a set of definitions and some equations. The equations are present to include equivalences such as $a + a = a$, with $a \in \Sigma$. The implementation for the nullable-function is shown in Section 3.2. Brzozowski derivatives are implemented in the same way the nullable function is implemented.

4.1 Automaton

In Section 2.4 an algorithmic method for creating a deterministic finite automaton (DFA) has been shown. Because the method is based on an algorithm it means that it can be implemented. For the implementation the first thing that is required is to define a representation of a DFA. A DFA is in its essence a set of states and a set of transitions. Therefore the result will be a list of states followed by a list of transitions as follows: `({state} ; ... ; {state} | [transition] ; ... ; [transition])`. A state is a regular expression which we are going to extend as we need the derivatives to make new states. A transition is defined as a triple `[r, a, s]` where `r` is the state we are moving from to a state `s`, while reading a symbol `a`. Here `r` may be equal to `s` to make a loop to the same state.

The challenge is now to create an automaton when a regular expression is given by the user. The method is identical to the method described in Section 2.4, which means calculating derivatives and introducing them as states if they are not already present. So the main challenges are: termination and making sure all transitions and states are created.

Termination can be guarenteed by ensuring a state that is already present in the list of states is not addded to that list. This requires us to make a helper function `in` which checks if a state is already in the current set of states. This will result in a rewrite law which will add a state (and the corresponding transition) to the automaton (which is 2 lists) if it is not already present. This operation needs to be present for all symbols $a \in \Sigma$ because we need to calculate all derivatives. This means that if we have a state r in the list we can add $r\prime$ to the list if $r\prime = \partial_a(r)$. This immediatly results in a transition from r to $r\prime$ while reading an a, `[r, a, r']`. In the code this addition looks as follows:

```
crl : (S ; Sl | Tl) => (D(S, a) ; S ; Sl | [S, a, D(S,a)] ; Tl) if in(D(S, a), S ; Sl) ==
false
```

What it does is the following: Given an automaton `(S ; Sl | Tl)` with a list of states `S ; Sl` (the `;`-symbol is concatenation of states) and a list of transtitions `Tl`. We can add the derivative of `S` with respect to `a` to the list `D(S, a) ; S ; Sl` and the corresponding transition `[S, a, D(S,a)] ; Tl`, if the state is not already present in the state list, that is, `in(D(S, a), S ; Sl) == false`.

One thing that is still missing now is the fact that sometimes we cannot add any new states from a state. This means that the derivative of that state is already present in the list. At this point we need a method of creating only a transition. The way to do this is to add a transition when from a state r to a state s when both of them are in the state list. Furthermore it needs to hold that $s = \partial_a(r)$ and there is no transition yet between between r and s. It can be argued that r must be equal to s because otherwise the transition would have been added when the state was added. Because Brzozowski has proven that a regular expressions has only finitely many derivatives modulo ACI it is ensured that this process will terminate, therefore no special termination operation is required. If we ignore helper functions the creation of a DFA only requires 4 rewrite laws, two rewrite laws for adding states and the corresponding transition, and another two are required as $|\Sigma| = 2$, we need seperate rules every $a \in \Sigma$. For the same reason two laws are required to add the transitions.

4.2 ProductAutomaton

The implementation of a product automaton is based on the implementation of DFA. This means the same representation as a DFA is used, with two changes. First of all a state is no longer a single regular expression, but a pair of regular expressions. Also a state is not only a pair of regular expressions, but the state will also indicate if the state is accepting. This means a state will now look as follows: {`r, s, 0`}, where `r` and `s` are the regular expressions. The third entry is either `0` or `1`; `1` means that the state is accepting while `0` indicates that the state is not accepting.

The addition of new states and transitions is based on the method described in Section 4.1. This time we work with pairs instead of regular expressions, so pairs need to be checked to be present in the state list instead of single regular expressions. Besides that check an important new check needs to be made. As explained in Section 2.5 it needs to be checked if $\nu(r) = \nu(s)$ so both regular expressions accept the empty word or neither

of them does so. To be more efficient we check $\nu(r) = \nu(s)$ immediately when we want to add the state. The other option is to just add all states and at the end check if the created automaton is valid. These are the main changes made to get from creating a DFA to creating a product automaton. Because we indicate whether a state is final or not (which we did not do in the creation of a DFA) the amount of rewrite laws required is doubled. It is doubled because we need the same laws both when the state is final and the state is not final. Because we optimized a little by checking if both regular expressions are accepting, we can immediatly stop if we find such a pair. To provide the user with error reporting, we than add a special state `ERROR` when this occurs. This yields the following code:

```
crl : (E1,E2,E3 ; Pl | Tl) => (ERROR ; E1,E2,E3 ; Pl | Tl) if v(D(E1, a)) =/= v(D(E2, a))
and in(ERROR, E1,E2,E3 ; Pl) == false .
```

The code is quite similar to the piece of code shown in the previous section. It adds the special `ERROR`-state to the automaton if the state to be added is not 'valid' ($\nu(r) = \nu(s)$). Furthermore we need to make sure to check that no `ERROR`-states are already present as otherwise multiple `ERROR`-sates are added. While this does not cause any problems, it is more 'elegant' if only one `ERROR`-state is added.

The code for creating a product automaton can be slightly altered to not check for equivalence but to check wether $L(r) \subseteq L(s)$. The first change is not to indicate which states are accepting, this would result in many rewrite laws. Many rewrite laws are not a problem but removing the indicator makes the code more readable. The only major change is that now not only states are considered 'valid' when $\nu(r) = \nu(s)$ but also when $\nu(s) = \varepsilon$. As long as the language of r does not accept any words that the language of s would not accept a state is valid. This does mean however that we need an extra rewrite law for all $a \in \Sigma$, which means a total of 8 rewrite laws.

4.3 Equivalence

As stated in Section 2.6, the goal is to more efficient in space. The first thing that we do to achieve this is no longer create transitions. These are not required to create all states, which determine whether two regular expressions are equivalent. Furthermore the transitions coming from a certain state can be found by calculating the derivative of both regular expressions and see what state it results in. The main challenge is now to find out what states are in the reflexive, symmetric and transitive closure, as well as check the $+$-closure. The way we check this is by calculating the closure from the state we are expanding and try and find the state we want to add in the closure of the state we started with. We illustrate this with an example.

Example 4.1 Closure calculation

Say we start with the following states:

$(r_1, r_3), (r_2, r_3), (r_4, r_5), (r_6, r_5), (r_1, r_4), (r_5, r_1)$

Of course, all of the r_x are regular expressions. Assume the derivative of (r_1, r_3) is (r_1, r_2), we need to check if (r_1, r_2) is in the closure. We use a secondary list; to this list we first add a state based on the state we are going to expand. Say this state is (r, s) then we add a state (r, r) to the new list. In this example we add (r_1, r_1) to the list. From this first

state we expand this list based on what is in the original list. If (r, r) is in the new list, we may add (r, t) to that list if (r, t) was in the original list. Furthermore we may add (t, r) if (t, r) was in the original list. For this example we get the following list:
(r_1, r_1)
(r_1, r_3)
(r_1, r_4)
(r_5, r_1)

Next we apply symmetry to these elements in the list. Which yields:
(r_1, r_1)
$(r_1, r_3), (r_3, r_1)$
$(r_1, r_4), (r_4, r_1)$
$(r_5, r_1), (r_1, r_5)$

This is then expended via transitivity:
(r_1, r_1)
$(r_1, r_3), (r_3, r_1)$
$(r_1, r_4), (r_4, r_1)$
$(r_5, r_1), (r_1, r_5)$
(r_5, r_5)

The final operation that is required is the +-closure. To make it more clear a pair a new example is made and only the pairs $\{r_2, r_6\}, \{r_3, r_4\}$ are present. these elements are present in the secondary litst which will be extended. Computing the +-closure modulo ACI we get:
$\{r_2, r_6\}, \{r_3, r_4\}$
$\{r_2 + r_3, r_6 + r_4\}$

These steps are repeated until the state we are searching for is found, as soon as that state is found we can stop extending and not add the state. Otherwise this process will terminate and the state is not found and we may add the extension of the original state to the automaton. In the implementation the closure calculation is not done in as systematic way as show in the example above. Maude works from top to bottom which means it will apply the equations and rewrite laws in a specific order. One thing to note is that reflexivity is not in this closure function. Reflexivity is dealt separately as follows: if a state (r, s) will be extended to (t, t) then we can stop extension of that new state as it is clear that $r \equiv s$. It is important of course to still check if $\nu(r) = \nu(s)$ and compute the other derivatives.

5 Experimental Results

For the results several 'variants' of the implementation are compared with different regular expressions. The abbreviation rew is short rewrite, which indicates the amount of rewrite that Maude had to do.

r	s	Product automaton with transitions	Product automaton without transitions	Equivalence
$(a+b)^* \cdot (ab) \cdot (a+b)^* + b^*a^*$	$(a+b)^*$	4 states, 11653 rew	4 states, 3594 rew	4 states, 2396 rew
$(a+b)^*$	$(a^* \cdot (a+b)^*)$	3 states, 2958 rew	3 states, 932 rew	2 states, 426 rew
$(a+b)^* \cdot b$	$(a+b)^* \cdot a$	1 state, 985 rew	1 state, 561 rew	? states, 400 rew
$(a+b) \cdot (a+b)^*$	$(a+b)^*$	0 states, 32 rew	0 states, 32 rew	0 states, 19 rew
$\varepsilon + \varepsilon$	$\varepsilon + \emptyset$	2 states, 776 rew	2 states, 117 rew	1 state, 46 rew
ab	ba	4 states, 2992 rew	4 states, 802 rew	? states, 279 rew
$b+b+b+a$	$a+a+a+b$	3 states, 1554 rew	3 states, 379 rew	1 state, 102 rew
$\varepsilon + ba$	$\varepsilon + ab$	0 states, 42 rew	0 states, 42 rew	0 states, 26 rew
$(a+b)^*$	$(a^* + (a+b)^*)$	2 states, 1294 rew	2 states, 496 rew	1 state, 152 rew
a^*	$a^* + \varepsilon$	3 states, 1585 rew	3 states, 486 rew	1 state, 87 rew
a^*a^*	a^*	3 states, 2238 rew	3 states, 725 rew	2 states, 326 rew
$(a+b)^*$	$(a^*b)^* \cdot a^*$	2 states, 3045 rew	2 states, 765 rew	2 states, 778 rew
$a(ba)^*$	$(ab)^*a$	3 states, 4539 rew	3 states, 853 rew	2 states, 500 rew

r	s	Only +-closure	Full equivalence without transitivity	Full equivalence
$(a+b)^* \cdot (ab) \cdot (a+b)^* + b^*a^*$	$(a+b)^*$	4 states, 2366 rew	4 states, 2396 rew	4 states, 2396 rew
$(a+b)^*$	$(a^* \cdot (a+b)^*)$	1 state, 192 rew	1 state, 196 rew	1 state, 196 rew
$(a+b)^* \cdot b$	$(a+b)^* \cdot a$	? states, 270 rew	? states, 754 rew	? states, 2059 rew
$(a+b) \cdot (a+b)^*$	$(a+b)^*$	0 states, 19 rew	0 states, 19 rew	0 states, 19 rew
$\varepsilon + \varepsilon$	$\varepsilon + \emptyset$	1 state, 42 rew	1 states, 46 rew	1 state, 46 rew
ab	ba	? states, 210 rew	? states, 315 rew	? states, 351 rew
$b+b+b+a$	$a+a+a+b$	1 state, 98 rew	1 state, 102 rew	1 state, 102 rew
$\varepsilon + ba$	$\varepsilon + ab$	0 states, 26 rew	0 states, 26 rew	0 states, 26 rew
$(a+b)^*$	$(a^* + (a+b)^*)$	1 states, 148 rew	1 state, 152 rew	1 state, 152 rew
a^*	$a^* + \varepsilon$	1 state, 83 rew	1 state, 87 rew	1 state, 114 rew
a^*a^*	a^*	2 states, 314 rew	1 state, 160 rew	1 state, 160 rew
$(a+b)^*$	$(a^*b)^* \cdot a^*$	2 states, 708 rew	2 states, 1276 rew	2 states, 2514 rew
$a(ba)^*$	$(ab)^*a$	2 states, 488 rew	2 states, 500 rew	2 states, 500 rew

The table above shows the results of several pairs regular expressions tested for equivalence on different parts of the implementation. The results show both the amount of states created by the method (this is shown as states). Secondly it shows how many rewrites are required to determine equivalence. While the goal is to minimize the number of states it is interesting to also look at the speed (less rewrites means faster) of the different implementations. In the table sometimes the number of states is unknown, which is indicated by a ?, this occurs in the variants when the regular expressions are not equivalent and an

empty list is returned. If the variant does not return the empty list, but produces an error state, then the amount of states shown is all states produced up until the error state was encountered. In both tables, the first two columns contain the two regular expressions which are checked for equivalence.

The third column in the first table is the results from the product automaton module. As described in Section 4 this module makes a product automaton based on the method in Section 2.5. The implementation, from which the results are shown in fourth column, also produces a product automaton however this variant does not produce any transitions. This is added to have a better comparison. The last column contains 'basic' equivalence; in equivalence the closure will be applied using reflexivity, symmetry and transitivity, buthis variant does not contain the +-closure. In the second table the third column contains the a variant in which we do check the closure, but only use +-closure. In the fourth column full equivalence is used, which is identical to equivalence but now the +-closure is also taken into account. However this is a variant in which transitivity is left out. The final variation is in the last cloumn of the second table here full equivalence is used without leaving anything out.

6 Conclusion and Discussion

The results from the previous section are interesting for several reasons. The first thing to notice is that in all test cases, and in any implementation variant the number of states is less than or equal to the method of a product automaton. That equivalence does not increase the number of states is obvious, that it decreases the number of states shows that the closure properties have a positive effect on the number of states. Equivalence also has the nice benefit of reducing the amount of rewrites required for most of the test cases.

An interesting result is gained from the pair $\{(a + b)^*\ ,\ (a^*b)^* \cdot a^*\}$. Here equivalence does not reduce the amount of states when comparing to the product automaton. It does however require about the same amount of rewrites, 3045 versus 2514. This is however an unfair comparison as equivalence does not have to use rewrites to generate transitions, while the product automaton does have to do this. It is therefore better to campare to the product automaton that does not produce transitions. Now it is 765 versus 2514 rewrites. Clearly in terms of speed the product automaton without transitions preforms better than equivalence does, and in terms of space they are equally good. It is interesting to also look at equivalence without transitivity. This is interesting as it requires significantly less rewrites when compared to equivalence (full equivalence to be exact), namely 1276 versus 2514 rewrites. This is still about 500 rewrites more than the product automaton without transitions. Of course the module for equivalence was designed to optimize in terms of space, but it is still interesting that it sometimes performs better while in other cases it does not.

If we are going to compare the full equivalence method to the full equivalence method without transitivity it shows that in none of the test cases the full equivalence method produces less states than it does without equivalence. Furthermore full equivalence performs worse in 2 cases in terms of rewrites. It is not clear why these two specific cases perform in this way, the first is a pair of expressions that are not equivalent while in the second they are not. This raises the question wether or not it is good to include transitivity to the implementation as it does not seem to reduce the amount of states. Neither does it improve the amount of rewrites, quite on the contrary it increases the number of rewrites required. Based on the results gained the 'best' variant would be full equivalence without transitivity. This variant yields a minimal number of states and required the least amount of rewrites to achieve this. If one would be looking at speed this would also be the best variant. All implementations are based in the goal to be most efficient in space, therefore it is not optimized in terms of speed. A implementation which has the goal to be as efficient as possible in speed would most likely out perform it.

For future work multiple things can be done. First of all more extensive testing could be done. This means more regular expressions and also regular expressions with a more extensive alphabet than only $\{a, b\}$. Secondly one would be intersted to extend the implementation to other types of automata. Here example could be, Mealy machines, Moore automata and weighted automata. It would also be nice to rewrite the automata in a format that can be used by a program that produces a graphical representation of an automata. An example of such a program would be graphviz. This topic has been researched more in the recent past an example of this can be found in [6].

Appendix

```
*** ********************************************************
*** Author: Leroy van Delft
*** This program will recognize if a given string is a
*** regular expression (RE) over the alphabet {a,b}.
*** 0 will be used for the empty language (or set).
*** 1 will be used for the empty string.
*** Extended with equations to reduce REs
*** Extended with the nullable opearation for REs
*** Extended with the Brzozowski Derivative
*** Extended with the checking if a String is part if a language
*** Extended with the ability to create a DFA
*** Extended with the checking if 2 REs describe the same language
*** Extended with the Derivative of a RE, over a RE
*** Extended with the checking if E1 is subset of E2
*** Extended equivalence checking with the closure
*** ********************************************************

*** This module contains the basic regular expression definitions,
*** aswell as basic quivalences among certain REs
mod RegularExpression is
  sorts RegExp Alphabet  .
  subsort Alphabet < RegExp .

  ops a
      b : -> Alphabet .
  ops 1
      0 : -> RegExp .

  *** Regular Expression related operations
  op _+_ : RegExp RegExp -> RegExp [ctor assoc comm idem prec 30] .
  op _&_ : RegExp RegExp -> RegExp [ctor assoc comm prec 25] .
  op _-_ : RegExp RegExp -> RegExp [ctor assoc prec 20] . *** Concatenation
  op _* : RegExp -> RegExp [ctor prec 10] .

  vars E : RegExp .
  vars A B : Alphabet .

  *** RE Equivalences
  eq 1 - E = E .
  eq 0 - E = 0 .
  eq 0 & 0 = 0 .
  eq 0 & E = 0 .
  eq E & E = E .
  eq 0 + 0 = 0 .
  eq 0 + E = E .
  eq E + E = E .
  eq (E *) * = E * .

endm

*** The module Nullable contains a function which can check wether a L(RE)
*** contains the empty word.
mod Nullable is
  protecting RegularExpression .
```

```
55
56    *** Function v determins wether a RegExp is nullable or not
57    op v : RegExp -> RegExp .
58
59    vars E E1 E2 : RegExp .
60    vars A B : Alphabet .
61
62    *** Nullable Equations
63    eq v(1) = 1 .
64    eq v(0) = 0 .
65    eq v(a) = 0 .
66    eq v(b) = 0 .
67    eq v(E1 - E2) = v(E1) & v(E2) .
68    eq v(E1 + E2) = v(E1) + v(E2) .
69    eq v(E *) = 1 .
70
71  endm
72
73  *** This module contains all functions related to derivatives of REs, which are:
74  *** - Brzozowski Derivatives
75  *** - Extended Brzozowski Derivatives for strings
76  *** - Derivatives over regular expressions
77  mod Derivative is
78    protecting RegularExpression .
79    protecting Nullable .
80
81    sorts String .
82    subsort Alphabet < String .
83
84    *** String related operations
85    op 1 : -> String .
86    *** We use [space] to concatenate 2 string symbols
87    op __ : Alphabet String -> String [id: 1] .
88    op D : RegExp Alphabet -> RegExp [ctor] . *** Derivative
89    op Ds : RegExp String -> RegExp [ctor] . *** Derivative for checking strings
90    op Dr : RegExp RegExp -> RegExp [ctor] . *** Derivative under regular expressions
91
92    vars E E1 E2 : RegExp .
93    vars A B : Alphabet .
94    var S : String .
95
96    *** Equations for Derivatives
97    *** Equations are used so they can be used by other modules
98    eq D(1, A) = 0 .
99    ceq D(B, A) = 1 if B == A .
100   ceq D(B, A) = 0 if B =/= A .
101   eq D(0, A) = 0 .
102   eq D(E *, A) = D(E, A) - E * .
103   eq D(E1 + E2, A) = D(E1, A) + D(E2, A) .
104   eq D(E1 - E2, A) = D(E1, A) - E2 + v(E1) - D(E2, A) .
105   eq D(E, 1) = E .
106
107   *** Derivatives for checking if a String is part of a Language
108   eq Ds(E, A S) = Ds(D(E, A), S) .
109   eq Ds(E, A) = Ds(D(E, A), 1) .
110   eq Ds(E, 1) = v(E) .
111
```

```
112   eq Dr(E, 0) = 0 .
113   eq Dr(E, 1) = E .
114   eq Dr(E, A) = D(E, A) .
115   eq Dr(E, (E1 + E2)) = Dr(E, E1) + Dr(E, E2) .
116   eq Dr(E, (E1 - E2)) = Dr(Dr(E, E1), E2) .
117   ceq Dr(E, (E1 *)) = E + Dr(Dr(E, E1), E1 *) if E =/= Dr(E, E1) .
118   ceq Dr(E, (E1 *)) = E if E == Dr(E, E1) .
119
120 endm
121
122 *** This module is used for the creation of a deterministic finite automaton
123 *** (DFA), for a given regular expression.
124 mod Automaton is
125   protecting RegularExpression .
126   protecting Nullable .
127   protecting Derivative .
128   protecting BOOL .
129
130   *** The declaration of the DFA 'building blocks'
131   sorts State Statelist Arrow Arrowlist Automaton .
132   subsort RegExp < State .
133   subsort State < Statelist .
134   subsort Arrow < Arrowlist .
135
136   *** Operations regarding DFAs
137   op nill : -> Statelist [ctor] .
138   op _;_ : State Statelist -> Statelist [comm assoc id: nill] .
139   op nill : -> Arrowlist [ctor] .
140   op [_,_,_] : State Alphabet State -> Arrow .
141   op _;_ : Arrow Arrowlist -> Arrowlist [comm assoc id: nill].
142   op (_|_) : Statelist Arrowlist -> Automaton .
143   op aut : RegExp -> Automaton .
144   op ins : State Statelist -> Bool .
145   op ins : Arrow Arrowlist -> Bool .
146
147   var E : RegExp .
148   vars S S2 : State .
149   var Sl : Statelist .
150   vars T T2 : Arrow .
151   var Tl : Arrowlist .
152
153   eq aut(E) = (E | nill) .
154
155   *** In: is a State or Transition part of the entire Transition or Statelist
156   eq ins(S, nill) = false .
157   eq ins(S, S ; Sl) = true .
158   ceq ins(S, S2 ; Sl) = ins(S, Sl) if S =/= S2 .
159
160   eq ins(T, nill) = false .
161   eq ins(T, T ; Tl) = true .
162   ceq ins(T, T2 ; Tl) = ins(T, Tl) if T =/= T2 .
163
164   crl [A-building1] : (S ; Sl | Tl) => (D(S, a) ; S ; Sl | [S, a, D(S,a)] ; Tl)
165     if ins(D(S, a), S ; Sl) == false .
166   crl [A-building2] : (S ; Sl | Tl) => (S ; Sl | [S, a, D(S,a)] ; Tl)
167     if (ins(D(S, a), S ; Sl) == true) and (ins([S, a, D(S,a)], Tl) == false) .
168
```

```
169   crl [B-building1] : (S ; Sl | Tl) => (D(S, b) ; S ; Sl | [S, b, D(S,b)] ; Tl)
170     if ins(D(S, b), S ; Sl) == false .
171   crl [B-building2] : (S ; Sl | Tl) => (S ; Sl | [S, b, D(S,b)] ; Tl)
172     if (ins(D(S, b), S ; Sl) == true) and (ins([S, b, D(S,b)], Tl) == false) .
173 endm
174
175 *** This module contains the checking if 2 RE accept the same language by means
176 *** of creating a product automaton. This also shows the user which states are final.
177 mod ProductAutomaton is
178   protecting RegularExpression .
179   protecting Nullable .
180   protecting Derivative .
181   protecting BOOL .
182
183   *** The declaration of the Product Automaton 'building blocks'
184   sorts Pair Pairlist Transition Transitionlist ProductAutomaton .
185   subsort Pair < Pairlist .
186   subsort Transition < Transitionlist .
187
188   *** The operations of the Product Automaton
189   op empty : -> Pairlist [ctor] .
190   op ERROR : -> Pair .
191   op _;_ : Pair Pairlist -> Pairlist [comm assoc id: empty] .
192   op {_,_,_} : RegExp RegExp RegExp -> Pair .
193   op empty : -> Transitionlist [ctor] .
194   op [_,_,_] : Pair Alphabet Pair -> Transition .
195   op _;_ : Transition Transitionlist -> Transitionlist [comm assoc id: empty].
196   op (_|_) : Pairlist Transitionlist -> ProductAutomaton .
197   op equal : Pair -> ProductAutomaton .
198   op in : Pair Pairlist -> Bool .
199   op in : Transition Transitionlist -> Bool .
200
201   vars E1 E2 : RegExp .
202   var E3 : RegExp .
203   vars P P1 P2 : Pair .
204   var Pl : Pairlist .
205   vars T T2 : Transition .
206   var Tl : Transitionlist .
207
208   ceq equal({E1,E2,E3}) = ({E1,E2,1} | empty) if (v(E1) == v(E2)) and v(E1) == 1 .
209   ceq equal({E1,E2,E3}) = ({E1,E2,0} | empty) if (v(E1) == v(E2)) and v(E1) == 0 .
210   ceq equal({E1,E2,E3}) = (ERROR | empty) if v(E1) =/= v(E2) .
211   eq in(P, empty) = false .
212   eq in(P, P ; Pl) = true .
213   ceq in(P1, P2 ; Pl) = in(P1, Pl) if P1 =/= P2 .
214
215   eq in(T, empty) = false .
216   eq in(T, T ; Tl) = true .
217   ceq in(T, T2 ; Tl) = in(T, Tl) if T =/= T2 .
218
219   *** All rewrite laws regarding the building of the Dual DFA
220   *** 1) The a-derivative of the 'current' state is not yet added and will
221   *** added now plus the corresponding transition
222   crl [Abuilding1] : ({E1,E2,E3} ; Pl | Tl) =>
223     ({D(E1, a),D(E2, a), 1} ; {E1,E2,E3} ; Pl |
224     [{E1,E2,E3}, a, {D(E1, a),D(E2, a),1}] ; Tl) if
225       ***(check if the a-derivative is not yet added)
```

```
      (in({D(E1, a),D(E2, a), 1}, {E1,E2,E3} ; Pl) == false) and
      ***(both expressions are nullable, and the nullable = 1)
      ((v(D(E1, a)) == v(D(E2, a))) and (v(D(E1,a)) == 1)) .
*** 2) The a-derivative of the 'current' state is already yet added
*** but a transition needs to be added
crl [Abuilding2] : ({E1,E2,E3} ; Pl | Tl) =>
  ({E1,E2,E3} ; Pl | [{E1,E2,E3}, a, {D(E1, a),D(E2, a),1}] ; Tl) if
    ***(check if the a-derivative is already added)
    (in({D(E1, a),D(E2, a), 1}, {E1,E2,E3} ; Pl) == true) and
    ***(check if the transition is already added)
    (in([{E1,E2,E3}, a, {D(E1, a),D(E2, a), 1}], Tl) == false) and
    ***(both expressions are nullable, and the nullable = 1)
    ((v(D(E1, a)) == v(D(E2, a))) and (v(D(E1,a)) == 1)) .

*** 3) The a-derivative of the 'current' state is not yet added and will
*** added now plus the corresponding transition
crl [Abuilding1] : ({E1,E2,E3} ; Pl | Tl) =>
  ({D(E1, a),D(E2, a), 0} ; {E1,E2,E3} ; Pl |
  [{E1,E2,E3}, a, {D(E1, a),D(E2, a),0}] ; Tl) if
    ***(check if the a-derivative is not yet added)
    (in({D(E1, a),D(E2, a), 0}, {E1,E2,E3} ; Pl) == false) and
    ***(both expressions are nullable, and the nullable = 0)
    ((v(D(E1,a)) == v(D(E2,a))) and (v(D(E1,a)) =/= 1)) .
*** 4) The a-derivative of the 'current' state is already yet added
*** but a transition needs to be added
crl [Abuilding2] : ({E1,E2,E3} ; Pl | Tl) =>
  ({E1,E2,E3} ; Pl | [{E1,E2,E3}, a, {D(E1, a),D(E2, a),0}] ; Tl) if
    ***(check if the a-derivative is already added)
    (in({D(E1, a),D(E2, a), 0}, {E1,E2,E3} ; Pl) == true) and
    ***(check if the transition is already added)
    (in([{E1,E2,E3}, a, {D(E1, a),D(E2, a), 0}], Tl) == false) and
    ***(both expressions are nullable, and the nullable = 0)
    ((v(D(E1,a)) == v(D(E2,a))) and (v(D(E1,a)) =/= 1)) .

*** 5) The b-derivative of the 'current' state is not yet added and
*** will added now plus the corresponding transition
crl [Abuilding1] : ({E1,E2,E3} ; Pl | Tl) =>
  ({D(E1, b),D(E2, b), 1} ; {E1,E2, E3} ; Pl |
  [{E1,E2,E3}, b, {D(E1, b),D(E2, b),1}] ; Tl) if
    ***(check if the b-derivative is not yet added)
    (in({D(E1, b),D(E2, b), 1}, {E1,E2,E3} ; Pl) == false) and
    ***(both expressions are nullable, and the nullable = 1)
    ((v(D(E1,b)) == v(D(E2,b))) and (v(D(E1,b)) == 1)) .
*** 6) The b-derivative of the 'current' state is already yet added
*** but a transition needs to be added
crl [Abuilding2] : ({E1,E2,E3} ; Pl | Tl) =>
  ({E1,E2,E3} ; Pl | [{E1,E2,E3}, b, {D(E1, b),D(E2, b),1}] ; Tl) if
    ***(check if the b-derivative is already added)
    (in({D(E1, b),D(E2, b), 1}, {E1,E2,E3} ; Pl) == true) and
    ***(check if the transition is already added)
    (in([{E1,E2,E3}, b, {D(E1, b),D(E2, b), 1}], Tl) == false) and
    ***(both expressions are nullable, and the nullable = 1)
    ((v(D(E1,b)) == v(D(E2,b))) and (v(D(E1,b)) == 1)) .

*** 7) The b-derivative of the 'current' state is not yet added and
*** will added now plus the corresponding transition
crl [Abuilding1] : ({E1,E2,E3} ; Pl | Tl) =>
```

```
283       ({D(E1, b),D(E2, b), 0} ; {E1,E2,E3} ; Pl |
284       [{E1,E2,E3}, b, {D(E1, b),D(E2, b),0}] ; Tl) if
285         ***(check if the b-derivative is not yet added)
286         (in({D(E1, b),D(E2, b), 0}, {E1,E2,E3} ; Pl) == false) and
287         ***(both expressions are nullable, and the nullable = 0)
288         ((v(D(E1,b)) == v(D(E2,b))) and (v(D(E1,b)) =/= 1)) .
289     *** 8) The b-derivative of the 'current' state is already yet
290     *** added but a transition needs to be added
291     crl [Abuilding2] : ({E1,E2,E3} ; Pl | Tl) =>
292       ({E1,E2,E3} ; Pl | [{E1,E2,E3}, b, {D(E1, b),D(E2, b),0}] ; Tl) if
293         ***(check if the b-derivative is already added)
294         (in({D(E1, b),D(E2, b), 0}, {E1,E2,E3} ; Pl) == true) and
295         ***(check if the transition is already added)
296         (in([{E1,E2,E3}, b, {D(E1, b),D(E2, b), 0}], Tl) == false) and
297         ***(both expressions are nullable, and the nullable = 0)
298         ((v(D(E1,b)) == v(D(E2,b))) and (v(D(E1,b)) =/= 1)) .
299
300     crl [nonequal] : ({E1,E2,E3} ; Pl | Tl) => (ERROR ; {E1,E2,E3} ; Pl | Tl) if
301       v(D(E1, a)) =/= v(D(E2, a)) and in(ERROR, {E1,E2,E3} ; Pl) == false .
302     crl [nonequal] : ({E1,E2,E3} ; Pl | Tl) => (ERROR ; {E1,E2,E3} ; Pl | Tl) if
303       v(D(E1, b)) =/= v(D(E2, b)) and in(ERROR, {E1,E2,E3} ; Pl) == false .
304
305 endm
306
307 *** This module will check if a regular language is a subset of a second regular language.
308 *** We check this using a product automaton based on the product automaton defined in the
309 *** module ProductAutomaton. Unlike the previous module this module does not show which
310 *** states are final.
311 mod Subset is
312   protecting RegularExpression .
313   protecting Nullable .
314   protecting Derivative .
315   protecting BOOL .
316
317   *** The declaration of the Product DFA 'building blocks'
318   sorts Pair Pairlist Transition Transitionlist SubsetAutomaton .
319   subsort Pair < Pairlist .
320   subsort Transition < Transitionlist .
321
322   *** The operations of the Product DFA
323   op empty : -> Pairlist [ctor] .
324   op _;_ : Pair Pairlist -> Pairlist [comm assoc id: empty] .
325   op {_,_} : RegExp RegExp -> Pair .
326   op empty : -> Transitionlist [ctor] .
327   op [_,_,_] : Pair Alphabet Pair -> Transition .
328   op _;_ : Transition Transitionlist -> Transitionlist [comm assoc id: empty] .
329   op (_|_) : Pairlist Transitionlist -> SubsetAutomaton .
330   op subset : Pair -> SubsetAutomaton .
331   op in : Pair Pairlist -> Bool .
332   op in : Transition Transitionlist -> Bool .
333
334   vars E1 E2 : RegExp .
335   vars P P1 P2 : Pair .
336   var Pl : Pairlist .
337   vars T T2 : Transition .
338   var Tl : Transitionlist .
339
```

```
340   ceq subset({E1,E2}) = ({E1,E2} | empty) if v(E1) == v(E2) .
341   ceq subset({E1,E2}) = ({E1,E2} | empty) if v(E2) == 1 and v(E1) == 0 .
342   ceq subset({E1,E2}) = (empty | empty) if v(E1) == 1 and v(E2) == 0 .
343   eq in(P, empty) = false .
344   eq in(P, P ; Pl) = true .
345   ceq in(P1, P2 ; Pl) = in(P1, Pl) if P1 =/= P2 .
346
347   eq in(T, empty) = false .
348   eq in(T, T ; Tl) = true .
349   ceq in(T, T2 ; Tl) = in(T, Tl) if T =/= T2 .
350
351   *** All rewrite laws regarding the building of the Subset DFA
352   crl [Abuilding1] : ({E1,E2} ; Pl | Tl) =>
353     ({D(E1, a),D(E2, a)} ; {E1,E2} ; Pl | [{E1,E2}, a, {D(E1, a),D(E2, a)}] ; Tl) if
354       (in({D(E1, a),D(E2, a)}, {E1,E2} ; Pl) == false) and
355       (v(D(E1, a)) == v(D(E2, a))) .
356   crl [Abuilding2] : ({E1,E2} ; Pl | Tl) =>
357     ({E1,E2} ; Pl | [{E1,E2}, a, {D(E1, a),D(E2, a)}] ; Tl) if
358       (in({D(E1, a),D(E2, a)}, {E1,E2} ; Pl) == true) and
359       (in([{E1,E2}, a, {D(E1, a),D(E2, a)}], Tl) == false) and
360       (v(D(E1, a)) == v(D(E2, a))) .
361
362   crl [Abuilding1] : ({E1,E2} ; Pl | Tl) =>
363     ({D(E1, a),D(E2, a)} ; {E1,E2} ; Pl | [{E1,E2}, a, {D(E1, a),D(E2, a)}] ; Tl) if
364       (in({D(E1, a),D(E2, a)}, {E1,E2} ; Pl) == false) and
365       ((v(D(E2, a)) == 1) and (v(D(E1, a)) == 0)) .
366   crl [Abuilding2] : ({E1,E2} ; Pl | Tl) =>
367     ({E1,E2} ; Pl | [{E1,E2}, a, {D(E1, a),D(E2, a)}] ; Tl) if
368       (in({D(E1, a),D(E2, a)}, {E1,E2} ; Pl) == true) and
369       (in([{E1,E2}, a, {D(E1, a),D(E2, a)}], Tl) == false) and
370       ((v(D(E2, a)) == 1) and (v(D(E1, a)) == 0)) .
371
372   crl [Bbuilding1] : ({E1,E2} ; Pl | Tl) =>
373     ({D(E1, b),D(E2, b)} ; {E1,E2} ; Pl | [{E1,E2}, b, {D(E1, b),D(E2, b)}] ; Tl) if
374       (in({D(E1, b),D(E2, b)}, {E1,E2} ; Pl) == false) and
375       (v(D(E1, b)) == v(D(E2, b))) .
376   crl [Bbuilding2] : ({E1,E2} ; Pl | Tl) =>
377     ({E1,E2} ; Pl | [{E1,E2}, b, {D(E1, b),D(E2, b)}] ; Tl) if
378       (in({D(E1, b),D(E2, b)}, {E1,E2} ; Pl) == true) and
379       (in([{E1,E2}, b, {D(E1, b),D(E2, b)}], Tl) == false) and
380       (v(D(E1, b)) == v(D(E2, b))) .
381
382   crl [Bbuilding1] : ({E1,E2} ; Pl | Tl) =>
383     ({D(E1, b),D(E2, b)} ; {E1,E2} ; Pl | [{E1,E2}, b, {D(E1, b),D(E2, b)}] ; Tl) if
384       (in({D(E1, b),D(E2, b)}, {E1,E2} ; Pl) == false) and
385       ((v(D(E2, b)) == 1) and (v(D(E1, b)) == 0)) .
386   crl [Bbuilding2] : ({E1,E2} ; Pl | Tl) =>
387     ({E1,E2} ; Pl | [{E1,E2}, b, {D(E1, b),D(E2, b)}] ; Tl) if
388       (in({D(E1, b),D(E2, b)}, {E1,E2} ; Pl) == true) and
389       (in([{E1,E2}, b, {D(E1, b),D(E2, b)}], Tl) == false) and
390       ((v(D(E2, b)) == 1) and (v(D(E1, b)) == 0)) .
391
392   crl [nonequal] : ({E1,E2} ; Pl | Tl) => (empty | empty) if
393     (v(D(E1, a)) == 1) and (v(D(E2, a)) == 0) .
394   crl [nonequal] : ({E1,E2} ; Pl | Tl) => (empty | empty) if
395     (v(D(E1, b)) == 1) and (v(D(E2, b)) == 0) .
396
```

```
endm

*** The module to which all previous modules have been working towards.
*** This module will check if 2 REs accept the same languages. While this
*** module is also based upon the creation of a product automaton. It used
*** transitivity, reflexivity and symmetry to check more efficiently.
*** The final result will not be a full functional product automaton.
*** To be more efficient in space transitions are no longer required, these
*** can be derived from the derivatives of a state.
mod Equivalence is
  protecting RegularExpression .
  protecting Nullable .
  protecting Derivative .
  protecting BOOL .

  *** The declaration of the Dual DFA 'building blocks'
  sorts Pair Pairlist Transition Transitionlist ProductAutomaton .
  subsort Pair < Pairlist .
  subsort Transition < Transitionlist .

  *** The operations of the Dual DFA
  op empty : -> Pairlist [ctor] .
  op TRUE : -> Pair .
  op FALSE : -> Pair .
  op ERROR : -> Pair [ctor] .
  op _;_ : Pair Pairlist -> Pairlist [comm assoc id: empty] .
  op {_,_} : RegExp RegExp -> Pair .
  op empty : -> Transitionlist [ctor] .
  op _;_ : Transition Transitionlist -> Transitionlist [comm assoc id: empty] .
  op (_|_) : Pairlist Transitionlist -> ProductAutomaton .
  op equal : Pair ->  ProductAutomaton .
  op in : Pair Pairlist -> Bool .
  op closure : Pair Pairlist Pairlist -> Bool .

  vars E1 E2 E3 E4 E5 E6 : RegExp .
  vars P P1 P2 : Pair .
  var Pl AltPl : Pairlist .
  var Tl : Transitionlist .

  ceq equal({E1,E2}) = ({E1,E2} | empty) if v(E1) == v(E2) .
  ceq equal({E1,E2}) = (FALSE | empty) if v(E1) =/= v(E2) .

  eq in(P, empty) = false .
  eq in(P, P ; Pl) = true .
  ceq in(P1, P2 ; Pl) = in(P1, Pl) if P1 =/= P2 .

  *** I use the fact that Maude works form top to bottom.
  *** By putting the return true operator at the top.
  ceq closure({E1,E2}, Pl, {E3,E4} ; AltPl) =
    true if in({E1,E2}, {E3,E4} ; AltPl) == true .
  ceq closure({E1,E2}, Pl, AltPl) =
    closure({E1,E2}, Pl, {E1,E1} ; AltPl) if in({E1,E1}, AltPl) == false .
  ceq closure({E1,E2}, {E3,E4} ; Pl, {E3,E5} ; AltPl) =
    closure({E1,E2}, {E3,E4} ; Pl, {E3,E4} ; {E3,E5} ; AltPl) if
    in({E3,E4}, {E3,E5} ; AltPl) == false .
  ceq closure({E1,E2}, {E3,E4} ; Pl, {E5,E4} ; AltPl) =
    closure({E1,E2}, {E3,E4} ; Pl, {E3,E5} ; {E5,E4} ; AltPl) if
```

```
    in({E3,E4}, {E5,E4} ; AltPl) == false .
  ceq closure({E1,E2}, Pl, {E3,E4} ; {E5, E6} ; AltPl) =
    closure({E1,E2}, Pl, {E3 + E5, E4 + E6} ; {E3, E4} ; {E5, E6} ; AltPl) if
    in({E3 + E5, E4 + E6}, {E3, E4} ; {E5, E6} ; AltPl) == false .
  ceq closure({E1,E2}, Pl, {E3,E4} ; AltPl) =
    closure({E1,E2}, Pl, {E4,E3} ; {E3,E4} ; AltPl) if
    in({E4,E3}, {E3,E4} ; AltPl) == false .
  ceq closure({E1,E2}, Pl, {E3,E4} ; {E4,E5} ; AltPl) =
    closure({E1,E2}, Pl, {E3,E5} ; {E3,E4} ; {E4,E5} ; AltPl) if
    in({E3,E5}, {E3,E4} ; {E4,E5} ; AltPl) == false .
  ceq closure({E1,E2}, Pl, {E3,E4} ; AltPl) = false if
    in({E1,E2}, AltPl) == false .

  *** All rewrite laws regarding the building of the Dual DFA
  *** 1) Add the a-derivative of {E1,E2} to the list of states, if all conditions are met.
  crl [Abuilding1] : ({E1,E2} ; Pl | Tl) => ({D(E1, a),D(E2, a)} ; {E1,E2} ; Pl | Tl) if
  v(D(E1, a)) == v(D(E2, a)) and ***(Are both REs final or non-final)
  (D(E1,a) =/= D(E2,a)) and ***(Reflexivity)
  ***(Check if the pair is in the symmetric, transitive and +-closure)
  (closure({D(E1, a),D(E2, a)} , {E1,E2} ; Pl , empty) == false) .

  *** 2) Add the b-derivative of {E1,E2} to the list of states, if all conditions are met.
  crl [Bbuilding1] : ({E1,E2} ; Pl | Tl) => ({D(E1, b),D(E2, b)} ; {E1,E2} ; Pl | Tl) if
  v(D(E1, b)) == v(D(E2, b)) and ***(Are both REs final or non-final)
  (D(E1,b) =/= D(E2,b)) and ***(/** Reflexivity */)
  ***(Check if the pair is in the symmetric, transitive and +-closure)
  (closure({D(E1, b),D(E2, b)} , {E1,E2} ; Pl, empty) == false) .

  crl [nonequal] : ({E1,E2} ; Pl | Tl) => (empty | Tl) if v(D(E1, a)) =/= v(D(E2, a)) .
  crl [nonequal] : ({E1,E2} ; Pl | Tl) => (empty | Tl) if v(D(E1, b)) =/= v(D(E2, b)) .
endm
```

References

[1] John C. Martin, Introduction to Languages and the Theory of Computation, Fourth edition, McGraw-Hill 2011

[2] Scott Owens, John Reppy, Aaron Turon, Regular-expression derivatives reexamined, Journal of Functional Programming, 2009

[3] Janusz A. Brzozowski, Derivatives of regular expressions, Journal of the ACM, 1964

[4] Maude, `http://maude.cs.uiuc.edu/`

[5] Theodore McCombs, Maude 2.0 Primer, August 2003

[6] Alexander Krause, Tobias Nipkow, Regular Expression Equivalence and Relational Algebra (not yet published), `http://www4.informatik.tu-muenchen.de/~krauss/papers/rexp.pdf`