

**UNIVERSITY OF PARIS SOUTH 11
UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE**

XML On - line Validation

(M.Sc. Thesis)

Dimitrios Kampas

*Paris
June 2009*

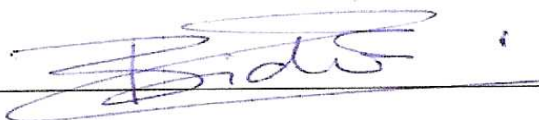
**DEPARTMENT OF COMPUTER SCIENCE
LRI, UNIVERSITY OF PARIS SOUTH 11
AND UNIVERSITY OF CRETE**

XML On – line Validation

Submitted in the 29th of June 2009 to the committee below
In partial fulfilment of the requirements for the degree of
Master in Computer Science
Under the common French-Greek graduate programme

Author: _____

Supervisor : _____



Nicole BIDOIT

Professor, University Paris South 11

Member : _____



Daniel ETIEMBLE

Professor, University Paris South 11

Grade : 16/20

Comments : Excellent

Université Paris Sud 11 - Orsay
Master Recherche Informatique
Bâtiment 490
91405 Orsay CEDEX

to all those who supported me...

XML on-line validation
Supervisors: Nicole Bidoit, Dario Colazzo

Dimitrios Kampas
Paris sud 11, Laboratoire de Recherche en Informatique

September 2, 2009

Contents

1	Introduction	7
1.1	Introduction	7
1.2	Related Work	8
1.3	Structure of the document	9
2	Preliminaries	11
2.1	Regular expression	11
2.2	Finite state machines	15
2.3	Conflict Free Regular Expression	16
3	Derivatives	19
3.1	RE derivatives	19
3.2	Using derivatives for membership checking	22
4	Conflict free RE Derivatives	25
4.1	Computing First	27
5	Complexity OF MCA	31
6	Derivatives and DFA construction	35
6.1	Automata construction algorithm(ACA)	36
7	DFA construction optimization	41
8	Conclusion and future work	47

Abstract

This work investigates the on-line validation of XML documents with respect to a DTD, under memory constraints. We consider a simple approach by examining the membership checking of a string with respect to a given regular expression. A DTD is studied in the form of a regular expression and an XML document is considered as a string. The membership checking is examined for a class of regular expressions that is called conflict free or single occurrence. It is shown that the majority of real worlds DTDs respect this restriction.

We use Brzozowski derivatives as an easy and efficient way to perform membership checking avoiding the automata construction. We provide an algorithm for membership checking on conflict free regular expressions. We examine the complexity of this algorithm on different classes of conflict free regular expressions and the condition, under which the complexity is linear, is provided .

The second approach is based on automata construction. Instead of computing the derivatives for each string on the fly, we precompute all the possible derivatives of the regular expression. We consider each derivative a state of an automaton recognizing the language described by the regular expression. The automata construction algorithm is provided and the problem of multiple derivations of same derivatives is examined.

Finally, we provide the basic notion of an optimization based on the symbol consuming during derivation. It enables to compute each derivative only once avoiding a considerable number of useless redundant derivations.

Chapter 1

Introduction

1.1 Introduction

Recently, the Extended Markup Language (XML) [14, 6, 8] has been considered as the standard for data exchange on the Web.

Many applications use XML documents. An XML document must be well-formed and well-structured. A well-formed XML document respects certain syntactic rules. However, those rules say nothing specific about the structure of the document. There are two ways of defining the structure of an XML document: DTD(Document Type Definition)[6, 8], the older and most restricted, and XML Schema[8], which offers extended possibilities.

In its most restrictive form, the problem of stream validation is to verify that an XML document is valid with respect to a given DTD in a single pass and using a fixed amount of memory, depending of the DTD but not on the size of the XML document. This is referred as *on-line validation* [15].

In this work we are concerned about the problem described above. We approach the problem of validation in two ways: by using finite-state automata and by avoiding the construction of FSA. To put the problem in perspective, note that validation with respect to a DTD amounts to checking membership of the tree associated with the XML document in a regular tree language. We focus on simple DTDs without constraints and the class of grammars considered are restricted(conflict-free).This restriction has been studied [10] in real worlds DTDs, and the result is that the majority of the rules satisfies the conflict free property. We study a DTD in the form of a regular expression. We use Brzozowski derivatives as introduced [4] in 1964

extended with the shuffle operator. Brzozowski derivatives are considered as an efficient way for constructing recognizers from a regular expression. We examine the efficiency of derivatives in membership checking [7] without automata construction [12]. We modify the automata construction algorithm introduced in [12] in order to be more effective on conflict free regular expressions. We give an optimized algorithm for automata construction avoiding a considerable number of derivative computation.

The main results of the master thesis is the exponential time complexity for membership checking without automata construction and the linear time complexity for conflict free regular expressions where kleene star nested depth is one.

1.2 Related Work

On-line validation of XML documents under memory constraints has been studied in [15]. The authors categorised the DTDs in classes and for each class they examined the automata construction. It is showed that for a class of DTDs named nonrecursive one can construct nondeterministic finite automata exponential in the size of the DTD for strong validation. For another class of automata, named recursive DTDs, it is showed that strong validation can be performed only by push down automata. In [14] the authors examine the class of DTDs that can be validated on-line using an FSA, these DTDs are called streamable. The hope of the authors was to prove that a DTD τ is streamable iff the set of trees accepted by the local-automaton for a DTD equals the set of trees valid for τ . In [5] DTDs are categorized as one-unambiguous or k-unambiguous and it is showed that for one-unambiguous and nonrecursive DTDs there is an algorithm for constructing deterministic automaton that validates documents with respect to a DTD. The size of the automaton is at most exponential with respect to the size of DTD. The authors gave the conditions under which a k-unambiguous and recursive DTD can be described by an one-counter automaton. In [1] the relations between regular expressions, finite state automata and derivatives have been investigated. In [12] the authors reexamine Brzozowski's work and reported new techniques for constructing scanner generators. In [9] Brzozowski's derivatives are extended in order to apply this technique to XSD, SGML and Relax NG [9, 11, 13]. The behavior of an extended form of

derivatives is studied and the problem of derivative explosion is examined in the case of regular expressions containing interleaving and Kleene-star. In [7] the authors introduced a different approach for membership checking to conflict free REs where Kleene star is only applied to symbol disjunctions. A linear time algorithm is presented which is based on the implicit representation of the constraints using a tree structure, and on a parallel verification of all constraints, using a residuation technique reminiscent Brzozowski's derivatives.

1.3 Structure of the document

The report is organized as follows: the basic notion of regular expressions, regular languages and finite state machines are reviewed in the preliminaries (chapter 2). In chapter 3, we introduce derivatives and we extend Brzozowski rules with shuffle. In chapter 4, we define a set of rules for conflict free regular expressions and we describe an algorithm for succeeding membership checking by using the rules defined. In the next chapter, we examine the complexity of the algorithm for different classes of regular expressions. In chapter 6, we investigate the approach based on automata construction. We provide the algorithm for automata construction using the derivative rules for conflict free regular expressions. In chapter 7, we introduce by examples an optimized algorithm for derivative calculations that eliminates the redundant calculation of derivatives.

Chapter 2

Preliminaries

We introduce here the basic formalism used throughout this paper. We also recall some basic notions related to regular languages and finite state machines.

2.1 Regular expression

Regular Expression

The syntax for regular expression which we present below includes concatenation, alternation and Kleene-star. Moreover, we include the empty set (\emptyset) and the shuffle and repetition operators.

Definition 2.1 (Regular Expression(RE)) *A regular expression over the alphabet Σ is defined as follows:*

$r, s ::=$	\emptyset	<i>emptyset</i>
	ϵ	<i>emptystring</i>
	α	$\alpha \in \Sigma$
	$r \cdot s$	<i>concatenation</i>
	$r s$	<i>alternation</i>
	$r \& s$	<i>interleaving</i>

| r^+ repetition

Table 2.1: *RE Syntax*

Remark 2.2 We use r^* to denote the kleene-star, where $r^* = r^+ \cup \{\epsilon\}$.

Note 2.3 We use \otimes to refer to any of $\{\cdot, |, \&\}$, when we need to specify common properties for them.

We use the notation $RE(\langle op_1 \rangle, \dots, \langle op_n \rangle)$ to refer to regular expressions using the operators $\langle op_1 \rangle, \dots, \langle op_n \rangle$.

For convenience we refer to $RE(\cdot, |, \&, *, +)$, as \overline{RE} .

Example 2.4 By $RE(\cdot, |)$, we refer to all regular expressions using the operators \cdot and $|$. The RE $r = (\alpha \cdot b) | c | (d \cdot e \cdot f)$ belongs to $RE(\cdot, |)$. On the contrary, $t = (\alpha \cdot b) \& (c \cdot d)$ does not belong to that set.

Below we give the definition of shuffle of two words and two languages.

Definition 2.5 (Shuffle or interleaving) The

shuffle set of two words $u, v \in \Sigma^*$, or two languages $\mathcal{L}_r, \mathcal{L}_s \subseteq \Sigma^*$ is defined as follows: $u \& v = \{u_1 \cdot v_1 \dots \cdot u_n \cdot v_n \mid u_1 \dots u_n = u, v_1 \dots v_n = v, u_i \in \Sigma^*, v_i \in \Sigma^*, n > 0\}$

$$\mathcal{L}_r \& \mathcal{L}_s = \bigcup_{u_1 \in \mathcal{L}_r, u_2 \in \mathcal{L}_s} (u_1) \& (u_2)$$

Example 2.6 $(ab) \& (XY)$ is a set consisting of the permutations of $abXY$ such that a comes before b and X comes before Y : $(ab) \& (XY) = \{abXY, aXbY, aXYb, XYab, XaYb, XabY\}$

Note 2.7 Shuffle is a very common operator used in XSD, relax NG and SGML regular expressions.

In relax NG the interleave pattern allows child elements to occur in any order. In the above example we present interleave pattern:

Example 2.8 This example would allow the card element to contain the name and email elements in any order:

```

<element name="addressBook">
<zeroOrMore>
  <element name="card">
    <interleave>
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </interleave>
  </element>
</zeroOrMore>
</element>

```

Example 2.9 Suppose now that we want to write a pattern for the HTML head element which requires exactly one title element, at most one base element and zero or more style, script, link and meta elements and suppose we are writing a grammar pattern that has one definition for each element. Then we could define the pattern for head as follows:

```

<define name="head">
  <element name="head">
    <interleave>
      <ref name="title"/>
      <optional>
        <ref name="base"/>
      </optional>
      <zeroOrMore>
        <ref name="style"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="script"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="link"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="meta"/>
      </zeroOrMore>
    </interleave>
  </element>
</define>

```

```

    </interleave>
  </element>
</define>

```

In SGML and XSD the shuffle operator is similar.

Regular Language

The regular languages are those that can be described by regular expressions according to the following definition.

Definition 2.10 (Regular language) *The language specified by a regular expression r is a set of strings $\mathcal{L}_r \subseteq \Sigma^*$ generated by the following rules:*

$$\begin{aligned}
 \mathcal{L}[\emptyset] &= \emptyset \\
 \mathcal{L}[\epsilon] &= \{\epsilon\} \\
 \mathcal{L}[\alpha] &= \{\alpha\} \\
 \mathcal{L}[r \cdot s] &= \{u \cdot v \mid u \in \mathcal{L}[r] \text{ and } v \in \mathcal{L}[s]\} \\
 \mathcal{L}[r \mid s] &= \mathcal{L}[r] \cup \mathcal{L}[s] \\
 \mathcal{L}[r \& s] &= \bigcup_{u_1 \in L_r, u_2 \in L_s} u_1 \& u_2 \\
 \mathcal{L}[r^+] &= \mathcal{L}[r \cdot r^*] \\
 \mathcal{L}[r^*] &= \{\epsilon\} \cup \mathcal{L}[r \cdot r^*]
 \end{aligned}$$

Table 2.2: Regular language

We define below the set of symbols of a RE or a string and the cardinality of the set as:

Definition 2.11 (Symbols of a regular expression) *$Sym(r)$ of any regular expression r is the set of all symbols appearing in r . $Sym(u)$ of any string u is the set of all symbols appearing in u .*

Note 2.12 *We denote as $|Sym(r)|$ the cardinality of $Sym(r)$.*

Example 2.13 *Suppose $r = \alpha \cdot (b \mid c) \cdot \alpha$, then $Sym(r) = \{\alpha, b, c\}$ and $|Sym(r)| = 3$.*

2.2 Finite state machines

Finite State Machine

A finite state machine or finite state automaton is an abstract machine that has a finite, constant amount of memory. Finite automata may operate on languages of finite words. It can be conceptualized as a directed graph where vertices are states and edges are transitions. There are a finite number of transitions and states. There is an input string that determines which transition is followed.

In deterministic automata (DFA) for each state there is at most one transition for each possible input. In non deterministic automata (NFA), there can be most than one transition from a given state for a given possible input. We are interested by deterministic automata.

Definition 2.14 (DFA) *A DFA over an alphabet Σ is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where:*

- a finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- a start state ($q_0 \in Q$)
- a set of accepting states ($F \subseteq Q$)

Let $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ be a DFA and $u = u_0 \cdot \dots \cdot u_{n-1}$ a string over the alphabet Σ . M accepts the string u if a sequence of states s_0, s_1, \dots, s_n exists in Q with the following conditions:

1. $s_0 = q_0$
2. $s_{i+1} = \delta(s_i, u_i)$, for $i = 0, \dots, n-1$
3. $s_n \in F$

As shown in the first condition, the machine starts in the start state q_0 . The second condition says that given each character of string u , the machine will transit from one state to another according to function δ . The last condition says that the machine accepts u if the last input symbol of

string u causes the machine to halt in one of the accepting states. Otherwise the string u is rejected.

The set of strings accepted by the DFA is formally defined as follows:

Definition 2.15 *A language accepted by a DFA is defined as the a set of strings $\{u \mid \delta(q_0, u) \in F\}$*

Example 2.16 Consider the regular language $1^*(0(1^*)0(1^*))^*$

- $Q = \{s_1, s_2\}$,
- $\Sigma = \{0, 1\}$,
- $q_0 = s_1$
- $F = \{s_1\}$,
- δ is defined as: $s_2 = (s_1, 0)$, $s_1 = (s_1, 1)$, $s_1 = (s_2, 0)$, $s_2 = (s_2, 1)$

The DFA is the one follows:

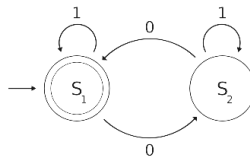


Figure 2.1: finite state automaton

2.3 Conflict Free Regular Expression

Conflict free regular expressions have been studied as duplicate-free [16, 10] or single occurrence [2, 3] regular expressions. Simply, it is the class of regular expressions for which each symbol appear only once in the regular expression. This restrictive class of regular expressions appear to be very common among the real worlds DTDs [10] . A formal definition is given below:

Definition 2.17 (Conflict Free RE(CF)) A regular expression $r = s^+$ is conflict free if the subexpression s is conflict free.

A regular expression $r = s \otimes t$ is conflict free if for each conflict free subexpression s, t , $Sym(s) \cap Sym(t) = \emptyset$.

Lemma 2.18 Let r be a conflict free $RE(\cdot, |, \&)$ and u a word in \mathcal{L}_r . Then each symbol in u occurs only once.

PROOF

Induction hypothesis

Let us assume that, if $u \in \mathcal{L}_r$ for some conflict free regular expression $r \in RE(\cdot, |, \&)$, where $|u| = n$, then each symbol in u occurs only once.

Let us consider a word $u_1 \in \mathcal{L}_{r_1}$ with:

- $|u_1| = n + 1$.
- r_1 is CF $RE(\cdot, |, \&)$.

Case 1: $r_1 = s_1 \cdot s_2$.

Assume that $u_1 = v_1 \cdot v_2$ where $v_i \in \mathcal{L}_{s_i}$ and s_i is a CF $RE(\cdot, |, \&)$.

By hypothesis: Lemma holds on s_1, v_1 and s_2, v_2 .

Because r_1 is CF we have that $Sym(s_1) \cap Sym(s_2) = \emptyset$.

This mean that:

Each symbol in u_1 occur once.

Case 2: $r_1 = s_1 | s_2$.

Then $u_1 \in \mathcal{L}_{s_1}$ or $u_1 \in \mathcal{L}_{s_2}$.

We have also that: By hypothesis: Lemma holds on s_1, u_1 or s_2, u_1 because $u_1 \in \mathcal{L}_{s_1}$ or $u_1 \in \mathcal{L}_{s_2}$. This means that:

Lemma holds for u_1, r_1 .

Case 3: $r_1 = s_1 \& s_2$.

Assume that $u_1 = v_1 \cdot v_2$, where $v_i \in \mathcal{L}_{s_i}$.

By hypothesis: Lemma holds on s_1, v_1 and s_2, v_2 . Because r_1 is CF we have that $Sym(s_1) \cap Sym(s_2) = \emptyset$. This means that:

Each symbol in u_1 occur once.

Lemma 2.19 For a given conflict free $RE(\cdot, |, \&)$ r , if a string $u \in \mathcal{L}_r$ then $|u| \leq n$, where $n = |Sym(r)|$.

PROOF

Immediate from 2.18

Chapter 3

Derivatives

Derivatives were presented by Brzozowski in 1964 as an elegant method to construct lexical recognizers. The concept of derivatives can be applied to any language. Intuitively the derivative of a language $\mathcal{L} \subseteq \Sigma^*$ with respect to a symbol $\alpha \in \Sigma$ is the language containing the suffix of strings belonging to \mathcal{L} and whose initial symbol is α .

Definition 3.1 (Derivative of a language) *The derivative of a language $\mathcal{L} \subseteq \Sigma^*$ with respect to a string u is defined to be $D_u\mathcal{L} = \{ v \mid u \cdot v \in \mathcal{L} \}$.*

Example 3.2 *Consider the language $\mathcal{L} = \{\alpha bb, \alpha\alpha bb, cd\}$. Then the derivative of the language with respect to α is the set: $D_\alpha(\mathcal{L}) = \{bb, \alpha bb\}$.*

The above language is described by the regular expression:

$r = \alpha \cdot (\alpha \cdot b \mid b) \cdot b \mid c \cdot d$. We can define the derivatives of a regular expression that describes the language \mathcal{L}_r , avoiding the calculation of derivatives of each element of the language. Below, the rules for regular expression calculation are given.

3.1 RE derivatives

Before the definition of derivatives is given, we need an intermediate function ν that says when a regular expression r is *nullable*.

Definition 3.3 (nullable) *A regular expression r is nullable iff $\epsilon \in \mathcal{L}_r$.*

The function ν below aims at identifying *nullable* regular expressions and is defined as follows:

$$\nu(r) = \begin{cases} \{\epsilon\} & \text{if } r \text{ is nullable} \\ \emptyset & \text{otherwise} \end{cases}$$

The function ν is alternatively defined by:

$$\begin{aligned} \nu(\emptyset) &= \emptyset \\ \nu(\epsilon) &= \{\epsilon\} \\ \nu(a) &= \emptyset \\ \nu(r \cdot s) &= \nu(r) \cap \nu(s) \\ \nu(r \mid s) &= \nu(r) \cup \nu(s) \\ \nu(r \ \& \ s) &= \nu(r) \cap \nu(s) \\ \nu(r^+) &= \emptyset \end{aligned}$$

Table 3.1: function ν

Remark 3.4 *The ν function of a kleene-stared RE r is : $\nu(r^*) = \{\epsilon\}$.*

Definition 3.5 (Brzowski's derivatives) *The definition of Brzowski's derivatives is shown in table 3.2:*

$$\begin{aligned} D_a(\emptyset) &= \emptyset, \\ D_a(\epsilon) &= \emptyset, \\ D_a(b) &= \begin{cases} \epsilon & \text{if } a = b, \\ \emptyset & \text{otherwise,} \end{cases} \\ D_a(r \cdot s) &= D_a(r) \cdot s \cup \nu(r) \cdot D_a(s), \\ D_a(r \mid s) &= D_a(r) \cup D_a(s), \\ D_a(r \ \& \ s) &= D_a(r) \ \& \ s \cup D_a(s) \ \& \ r, \end{aligned}$$

$$D_a(r^+) = D_a(r) \cdot r^*,$$

Table 3.2: Brzozowski's derivatives

Remark 3.6 *The derivative of a RE r under Kleene-star is: $D_a(r^*) = D_a(r) \cdot r^*$.*

Example 3.7 *Consider the language that is described by the regular expression $r=(\alpha \mid b)c$ (we eliminate \cdot before c for convenience). Then the derivative of r with respect to α is c . The derivative of r with respect to b is c as well. And the derivative of r with respect to c is the empty set.*

Note 3.8 *We use the term residual RE or simply residual to refer to a derivative of a RE.*

Lemma 3.9 *The derivative of a language \mathcal{L}_r with respect to a symbol α and the semantics of the derivative of the language \mathcal{L}_r commute. Formally: $\mathcal{L}_{D_\alpha(r)} = D_\alpha(\mathcal{L}_r)$.*

PROOF. In case that $r = \emptyset$ or ϵ . The above is true.

We will prove that both languages are described by the same automaton. Suppose that the regular expression r defines a language \mathcal{L}_r described by an automaton $\langle Q, \Sigma, T, q_0, F \rangle$.

Then, $r_1 = D_\alpha(r)$, the regular expression r_1 defines the language \mathcal{L}_{r_1} , which contains the strings u such that $\alpha.u$ belongs to \mathcal{L}_r . Hence \mathcal{L}_{r_1} is given by the automaton $\langle Q, \Sigma, T, \delta(q_0, \alpha), F \rangle$ (1).

The language \mathcal{L}_r is described by the automaton $\langle Q, \Sigma, T, q_0, F \rangle$. The derivative of \mathcal{L}_r , $D_\alpha(\mathcal{L}_r)$ with respect to a symbol α is L_1 . By definition 3.1 the automaton describing L_1 is the automaton with initial state $\delta(q_0, \alpha)$. The automaton is $\langle Q, \Sigma, T, \delta(q_0, \alpha), F \rangle$ (2). We see that both are described by the same automaton.

3.2 Using derivatives for membership checking

Suppose we want to check whether a string u is contained in the language defined by r . Formally, we want to check if $u \in L_r$. We have $u \in L_r$ if and only if $\epsilon \in L_{D_u(r)}$ which is true exactly when $\nu(D_u(r)) = \{\epsilon\}$.

Combining the above leads to an algorithm for testing the membership of a string to the language defined by a given regular expression. We will use the symbol \vdash to express the relation $r \vdash u$ (**u is valid with respect to r**) defined as:

$$\begin{aligned} r \vdash \epsilon &\Leftrightarrow \nu(r) = \{ \epsilon \} \\ r \vdash a \cdot u &\Leftrightarrow D_a(r) \vdash u \end{aligned}$$

Membership checking algorithm (MCA)

MCA computes a derivative of a regular expression for each symbol in the string in order to check the membership of the string. If the final residual of the regular expression is nullable, then $u \in r$. MCA is defined as follows:

Definition 3.10 (MCA) *MCA takes as input a RE r and a string u and returns true if $u \in r$. We define MCA as follows:*

```

Algorithm MCA
Input: RE r, string u
Output: boolean

while  $u \neq \epsilon$  and  $r \neq \emptyset$ 
    assuming  $u$  is  $\alpha \cdot u_1$ 
         $r \leftarrow D_\alpha(r)$  ;
         $u \leftarrow u_1$  ;
end-while
if  $\nu(r) = \epsilon$  then return true else return false

```

Table 3.3: MCA

Example 3.11 Suppose $r = \alpha \cdot c \cdot b^*$ and $u = \alpha cbb$. We want to check if u is valid with respect to r .

$$\begin{aligned}
 & a \cdot c \cdot b^* \vdash acbb \\
 & D_a(a) \cdot c \cdot b^* \cup \nu(\alpha) \cdot D_\alpha(c \cdot b^*) \vdash cbb \\
 & \quad c \cdot b^* \cup \emptyset \vdash cbb \\
 & D_c(c) \cdot b^* \cup \nu(c) \cdot D_c(b^*) \vdash bb \\
 & \quad b^* \cup \emptyset \vdash bb \\
 & D_b(b^*) \cup \nu(b^*) \cdot D_b(b^*) \vdash b \\
 & \quad b^* \cup \emptyset \vdash b \\
 & D_b(b^*) \cup \nu(b^*) \cdot D_b(b^*) \vdash \epsilon \\
 & \quad b^* \cup \emptyset \vdash \epsilon \\
 & \nu(b^*) = \{\epsilon\}(true)
 \end{aligned}$$

When a derivative doesn't match the string, we reach a derivative that is \emptyset , and stop.

Chapter 4

Conflict free RE Derivatives

For conflict free regular expressions, taking advantage of the unique occurrence of each symbol in the regular expression, we could define the derivative in such a way to avoid the calculation of a number of derivatives that lead to empty set (example 3.9) during the MCA. We are interested to know a priori the set of symbols which are potentially relevant to the computation of the derivatives. For that purpose we define the *First set* as follows:

Definition 4.1 (First set) *The First set of a regular expression r is defined as: $\text{First}(r) = \{\alpha \in \text{Sym}(r) \mid \exists u: \alpha u \in \mathcal{L}_r\}$*

$\text{First}(r)$ represents the set of first symbols of words in L_r .

In order to calculate the First set of a regular expression r , we define a function F . **Definition** of $F(r)$ is given by following rules:

$$\begin{aligned} F(\emptyset) &= F(\epsilon) = \emptyset \\ F(\alpha) &= \{\alpha\}, \text{ for any } \alpha \\ F(r \mid s) &= F(r) \cup F(s), \\ F(r \cdot s) &= \begin{cases} F(r), & \text{if } \epsilon \notin \mathcal{L}(r) \\ F(r) \cup F(s), & \text{if } \epsilon \in \mathcal{L}(r) \end{cases} \\ F(r \&x s) &= F(r) \cup F(s) \\ F(r^+) &= F(r) \end{aligned}$$

Table 4.1: Function F

We use the *First set* to rewrite the derivative rules shown in table 3.2 for conflict free regular expressions, taking advantage of the single occurrence restriction of the RE.

Definition 4.2 (Conflict free RE derivatives) *The derivatives for a CF regular expression with respect to a symbol α are defined as follows:*

$$\begin{aligned}
 D_a(\emptyset) &= \emptyset, \\
 D_a(\epsilon) &= \emptyset, \\
 D_a(b) &= \begin{cases} \epsilon & \text{if } a = b, \\ \emptyset & \text{otherwise,} \end{cases} \\
 D_a(r \cdot s) &= \begin{cases} D_a(r) \cdot s & \text{if } a \in \text{First}(r), \\ D_a(s) & \text{if } a \in \text{First}(s) \text{ and } \epsilon \in L(r), \\ \emptyset & \text{otherwise,} \end{cases} \\
 D_a(r \mid s) &= \begin{cases} D_a(r) & \text{if } a \in \text{First}(r), \\ D_a(s) & \text{if } a \in \text{First}(s), \\ \emptyset & \text{otherwise,} \end{cases} \\
 D_a(r \& s) &= \begin{cases} D_a(r) \& s & \text{if } a \in \text{First}(r), \\ D_a(s) \& r & \text{if } a \in \text{First}(s), \\ \emptyset & \text{otherwise} \end{cases} \\
 D_a(r^+) &= \begin{cases} D_a(r) \cdot r^* & \text{if } a \in \text{First}(r), \\ \emptyset & \text{otherwise,} \end{cases}
 \end{aligned}$$

Table 4.2: CF regular expression Derivatives

Remark 4.3 *The derivative of a conflict free RE under Kleene-star is r:*

$$D_a(r^*) = \begin{cases} D_a(r) \cdot r^* & \text{if } a \in \text{First}(r), \\ \emptyset & \text{otherwise} \end{cases}$$

Lemma 4.4 *The Brzowski derivative rules shown in table 3.2 are equivalent with the derivative rules for Conflict free RE shown in table 4.2.*

PROOF. For the shuffle operator the rules is:

$$D_\alpha(r \& s) = D_\alpha(r) \& s \cup D_\alpha(s) \& r.$$

Suppose that there is a conflict free regular expression $t = r \& s$.

Case 1: $\alpha \in \text{First}(r)$, then $\alpha \notin \text{Sym}(s)$. So,

$$D_\alpha(s) = \emptyset, \text{ Hence } D_\alpha(t) = D_\alpha(r) \& s.$$

Case 2: $\alpha \in \text{First}(s)$, then $\alpha \notin \text{Sym}(r)$. So,

$$D_\alpha(r) = \emptyset, \text{ Hence } D_\alpha(t) = D_\alpha(s) \& r.$$

Case 3: $\alpha \notin \text{Sym}(t)$, then $\alpha \notin \text{Sym}(s)$ and $\alpha \notin \text{Sym}(r)$.

Hence, $D_\alpha(t) = \emptyset$.

The proof proceed in a similar manner for the other rules.

4.1 Computing First

It is worth mentioning that it is not necessary to calculate the *First set* for each residual regular expression during the MCA. We need to process once the initial regular expression tree assigning to each symbol an id that facilitate the *First set* calculation for every residual expression during the derivation process.

At the implementation level, we could define a function that assigns an id to each symbol. Intuitively, id identifies which symbols of a residual regular expression are in the First set. The symbols contained in the First set, are marked with the same id.

In order to compute the id for each symbol in the RE, we construct the RE tree and we apply on it the recursive function $\text{recid}(r, \text{id}) \rightarrow \text{id}$. We construct as many sets as the ids and in each set are contained the symbols with the same id.

Definition 4.5 (id) *We define id as follows:*

Input: a RE r , id

Output: id

$recid(r, n) =$

if $r = \epsilon$ **then** return $n - 1$

if $r = \alpha$ **then** $id(\alpha) = n$

if $r = r_1 \cdot r_2$ **then** return $recid(r_2, recid(r_1, n) + 1)$

if $r_1 | r_2$ **then** return $\max \{recid(r_1, n), recid(r_2, n)\}$

if $r = r_1^+$ **then** return $recid(r_1, n)$

Table 4.3: Recid function

Note 4.6 We rewrite every atom RE α^* as: $(\alpha^+ | \epsilon)$.

The id initial value is 1.

Example 4.7 Consider the regular expression:

$r = \alpha b c^*(d e f^* | g h) x y$. We want to calculate the $First(r)$.

We rewrite the RE $r = \alpha b (c^+ | \epsilon) (d e (f^+ | \epsilon) | g h) x y$. We apply the $recid$ on r . The result is the one shown below:

$r = \alpha_1 b_2 c_3^*(d_3 e_4 f_5^* | g_3 h_4) x_5 y_6$

Note 4.8 Due to the fact that the id is subtracted by one every time ϵ is encountered, we have that $id(c) = id(d) = 3$ and $id(f) = id(x) = 5$.

Then we construct the above sets:

$First_{id=1} = \{\alpha\}$, $First_{id=2} = \{b\}$, $First_{id=3} = \{c, d, g\}$,

$First_{id=4} = \{e, f\}$, $First_{id=5} = \{f, x\}$, $First_{id=6} = \{y\}$.

So during the derivation process we check the id of the initial symbol of the residual RE and we choose the $First$ set with equal id .

For the regular expression r where the $id(\alpha) = 1$, the $First$ set is the $First_{id=1}$.

If we derive wrt α we have:

$r_1 = D_\alpha(r) = b_2 c_3^*(d_3 e_4 f_5^* | g_3 h_4) x_5 y_6$.

We go on checking the $id(b) = 2$. The $First$ set is the $First_{id=2}$. If we derive wrt to b we have:

$$r_2 = D_b(r) = c_3^*(d_3e_4f_5^* \mid g_3h_4)x_5y_6$$

The $id(c) = 3$, hence the First set is $First_{id=3}$ and so on.

Chapter 5

Complexity OF MCA

In this section we examine the time and space complexity of MCA for conflict free REs. For that purpose we first define the depth of repetition for a regular expression.

Definition 5.1 (Depth of repetition) *For a given regular expression r , we define as depth of repetition $\mathbf{dept}(r)$ of r the maximum number of nested kleene-closure or repetition symbols in a regular expression.*

$\mathit{dept}(\emptyset)$	$=$	0
$\mathit{dept}(\epsilon)$	$=$	0
$\mathit{dept}(b)$	$=$	0
$\mathit{dept}(r \otimes s)$	$=$	$\max(\mathit{dept}(r), \mathit{dept}(s))$
$\mathit{dept}(r^+)$	$=$	$\mathit{dept}(r) + 1$
$\mathit{dept}(r^*)$	$=$	$\mathit{dept}(r) + 1$

Table 5.1: *Depth*

Example 5.2 *Consider the regular expression: $r = (a \cdot b \mid c^+)^*$. The depth of repetition $\mathbf{dept}(r)$ of r is 2.*

Intuitively, the following lemma states that for a conflict free RE r without any of the repetition operators $\{+, *\}$, the symbol with respect of which we derive r is not contained in the residual RE. Formally:

Lemma 5.3 *For a given conflict free $RE(\cdot, |, \&)$ r , the derivative $s = D_\alpha(r)$ of r with respect to a symbol α is such that $\alpha \notin \text{Sym}(s)$.*

PROOF

If $r = \emptyset$. Lemma holds.

Suppose $r \neq \emptyset$ a conflict free $RE(\cdot, |, \&)$, and $\alpha \in \text{Sym}(s)$. Then $\exists u \in \mathcal{L}_s$ such that α occurs in u . Thus we have that, α occurs twice in $\alpha \cdot u$, Absurd by lemma 2.18.

Hence $\alpha \notin \text{Sym}(s)$.

Theorems 5.5, 5.7 and 5.10 provide the complexity of MCA for different classes of REs.

The lemma below states the linear complexity of membership checking for a CF regular expression with no repetition operators. Formally:

Theorem 5.4 *The time complexity of checking the membership of a string u with respect to a CF $RE(\cdot, |, \&)$ r is $O(n)$, where $n = |\text{Sym}(r)|$.*

PROOF

There are two cases:

1. If $u \notin \mathcal{L}_r$.

- If $|\text{Sym}(r)| \leq |u|$.

Let $u = \alpha_1 \cdots \alpha_m$, where $m \geq n$. Then $D_{\alpha_1 \cdots \alpha_n}(r) = \emptyset$, because after n derivation r has no more symbols. The complexity is: $O(n + 1) \sim O(n)$.

- If $|u| \leq |\text{Sym}(r)|$.

Let $u = \alpha_1 \cdots \alpha_m$, $m \leq n$. Then $D_{\alpha_1 \cdots \alpha_m}(r) = \acute{r}$. Because after m derivations there are no more symbols that derive in u , the complexity is: $O(m) \leq O(n)$.

2. If $u \in \mathcal{L}_r$.

Let $u = \alpha_1 \cdots \alpha_m$. Then by lemma 5.4: $D_{\alpha_1 \cdots \alpha_m}(r) = \epsilon$. After m steps there no more symbols in u to derive. Hence the complexity is:

$O(m) \leq O(n)$.

Above we propose a way to calculate the number of derivations for checking the membership of a string u with respect a given RE r . The number of

derivations $N(u, r)$ is the summary of the derivations of each symbol of the string u in the regular expression s that it is derived. Formally:

Proposition 5.5 *For a given string $u = \alpha_1 \cdots \alpha_n$ and a RE r . The number of derivations for computing $D_u(r)$ is:*

$$N(u, r) = \sum_{i=1}^{|u|} N(\alpha_i, D_{\alpha_1 \cdots \alpha_{i-1}}(r)), \text{ where}$$

$$N(\alpha, r) = \begin{cases} 1 & \text{if } dept(r) = 0, \\ dept(r) & \text{if } dept(r) > 0 \end{cases}$$

The following lemma denotes that the complexity remains linear in the case of a conflict free RE r with $dept(r) = 1$.

Theorem 5.6 *The time complexity of checking the membership of a string u with respect to a CF RE r of $dept(r) = 1$, is $O(m)$, where $m = |u|$.*

PROOF

We assume that $u = \alpha_1 \cdots \alpha_m$. There are two cases:

1. $u \in \mathcal{L}_r$. Then each residual RE s during derivation process has $dept(s) \leq 1$, and $D_u(r) = \epsilon$. So by proposition 5.6 we have: $N(u, r) = \underbrace{1 + \cdots + 1}_m = m$.
2. $u \notin \mathcal{L}_r$.
 - If $D_{\alpha_1 \cdots \alpha_p}(r) = \emptyset$, for some $p \leq m$.
Then $N(u, r) = p$. Hence the time complexity is $O(m)$.
 - If $D_u(r) = \acute{}$.
Then after m derivations there are no more symbols to derive in u . So $N(u, r) = m$. Hence the complexity is $O(m)$.

Hence the time complexity is $O(m)$.

Lemma 5.7 *The time complexity of checking the membership of a string u w.r.t a CF RE $r = ((r_1)^* \otimes r_2)^* \otimes \cdots \otimes r_{k-1})^*$ is $O(k^m)$, where $r_i = \alpha_1^* \cdots \alpha_k^*$, $dept(r) = k$ and $m = |u|$.*

Lets see a small example before we go on with the proof.

Example 5.8 *Consider the RE $r = ((\alpha^* b^* | c^*)^*)^*$ and a string $u = \alpha \alpha$ for which we want to check if $u \in \mathcal{L}_r$.*

We have to derive twice with respect to α . The derivative of r with respect to α is:

$$r_1 = D_\alpha(r) = \alpha^* b^* \cdot (\alpha^* b^* | c^*)^* \cdot ((\alpha^* b^* | c^*)^*)^* \text{ and} \\ D_\alpha(r_1) = \alpha^* b^* | \alpha^* b^* \cdot (\alpha^* b^* | c^*)^* | \alpha^* b^* \cdot (\alpha^* b^* | c^*)^* \cdot ((\alpha^* b^* | c^*)^*)^*.$$

In this example we notice the exponential behavior of the regular expression after two derivations.

PROOF

If we derive with respect to a symbol $\alpha \in \text{Sym}(r)$, due to the rule $D_\alpha(r^*) = D_\alpha(r) \cdot r^*$, we have that $D_\alpha(r) = \underbrace{D_\alpha(r_1)}_{s_1} \cdot \underbrace{r_1^*}_{s_2} \cdot \underbrace{((r_1)^* \otimes r_2)^*}_{s_3} \cdots \underbrace{r}_{s_k}$.

It is obvious that $D_\alpha(r)$ is not conflict free and $\nu(s_i) = \{\epsilon\}$.

So if we go on the derivation process w.r.t. to a symbol $b \in \text{Sym}(r_1)$, we have to apply the rule:

$$D_\alpha(r \cdot s) = D_\alpha(r) \cdot s \cup \nu(r) \cdot D_\alpha(s), \text{ where } r = D_\alpha(r_1) \text{ and } s = r_1^* \cdot ((r_1)^* \otimes r_2)^* \cdots r.$$

$$\text{So, } D_{b\alpha}(r) = \underbrace{D_{b\alpha}(r_1)}_{q_1} | \underbrace{D_b(r_1)}_{q_1} \cdot \underbrace{r_1^*}_{q_2} | \underbrace{D_b((r_1)^* \otimes r_2)}_{q_1} \cdot \underbrace{((r_1)^* \otimes r_2)^*}_{q_2} | \cdots \\ | \underbrace{D_b((r_1)^* \otimes r_2)}_{q_1} \cdot \underbrace{((r_1)^* \otimes r_2)^*}_{q_2} \cdot \underbrace{(((r_1)^* \otimes r_2)^*)^*}_{q_3} \cdots \underbrace{r}_{q_k} \\ \underbrace{\hspace{15em}}_{t_k}$$

It is obvious that the form of $D_{b\alpha}(r) = \bigcup_{i=1}^k t_i$, where $t_i = \bullet q_i$ and $1 \leq i \leq \text{dept}(s_i)$, by $\bullet q_i$ we represent the concatenation of one or more regular expressions.

In order to calculate the number of derivations for $D_u(r)$, we consider that

$$\text{dept}(s_i) \text{ is bounded by } k. \text{ Hence } N(u, r) \leq \sum_{i=1}^{|u|} k^i \Rightarrow N(u, r) \leq \frac{k(k^m - 1)}{k - 1}.$$

Hence the complexity is $O(k^m)$.

Theorem 5.9 *The complexity of checking the membership of a string u w.r.t. a CF RE r is $O(k^m)$, where $k = \text{dept}(r)$ and $|u| = m$.*

PROOF

By lemma 5.8, we have that for the subset of conflict free regular expressions of the form: $r = (((r_1)^* \otimes r_2)^* \otimes \cdots \otimes r_{k-1})^*$ the complexity of checking the membership of the string u w.r.t. r is $O(k^m)$. Hence, in the general case of a conflict free regular expression r , the complexity is $O(k^m)$.

Chapter 6

Derivatives and DFA construction

The MCA decides if a string is valid with respect to a given RE. If we consider a regular expression as a DTD and a given string as an XML document then, we claim that MCA is suitable for on-line XML documents validation with respect to a given DTD. The disadvantage of this algorithm is that for every given XML document we have to check its validity by calculating the derivatives for each symbol of the XML document. If, instead of computing the derivatives on the fly, we precompute the derivative for each symbol in Σ , we can construct a DFA recognizing the language of the RE. In [12], an algorithm is presented that builds an automata using the derivatives. In this section we present a different version of this algorithm using the set First.

Before we present the algorithm for automata construction by using derivatives, we define the notion of equality of two REs.

Definition 6.1 (Equivalence of regular expressions) *Two regular expressions r, s are equivalent when they describe the same languages. Formally: $r \equiv s$ iff $\mathcal{L}_r = \mathcal{L}_s$.*

Example 6.2 *Let $r_1 = (\alpha|b^*)^*$ and $r_2 = (\alpha|b)^*$, then $r_1 \equiv r_2$ because $\mathcal{L}_{r_1} = \mathcal{L}_{r_2}$.*

```

Algorithm ACA
Input: a RE r
Output:  $\langle Q, \Sigma, \delta, q_0, F \rangle$ 

trans q ( $\alpha, (Q, \delta)$ ) =
let  $D_\alpha(q) = q_\alpha$ 
if  $\exists q_1 \in Q$  such that  $q_1 \equiv q_\alpha$ 
     $\{(Q, \delta \cup (q, \alpha) \mapsto q_1)\}$ 
else
    let  $Q_1 = Q \cup \{q_1\}$ 
    let  $\delta_1 = \delta \cup \{Q_1, \delta_1, q_\alpha\}$ 
    in collect ( $Q_1, \delta_1, q_\alpha$ )

collect( $Q, \delta, q$ ) = enclose(trans q)( $Q, \delta$ ) F(r)

constrDFA r =
let  $q_0 = D_\epsilon(r)$ 
let  $(Q, \delta) = \text{collect}(\{q_0\}. \{\}, q_0)$ 
let  $F = \{q \mid q \in Q \text{ and } \nu q = \epsilon\}$ 
in  $\langle Q, \Sigma, \delta, q_0, F \rangle$ 

```

Table 6.1: ACA

6.1 Automata construction algorithm(ACA)

Table 10 gives the algorithm for Constructing a DFA

$\langle Q, \Sigma, \delta, q_0, F \rangle$ using derivatives.

The function *trans* constructs the transition from a state q when a symbol α is encountered.

The function *collect* collects all the possible transitions from the state q .

The function *constrDFA* constructs the DFA.

The DFA construction algorithm as it is presented in table 10 introduces a new state when no equivalence state is present. Brzozowski proved that this check for state equivalence guarantees the minimality of the DFA produced by ACA, but checking equivalence is expensive. In practice we weaken the

notion of equality to similarity.

Definition 6.3 Let \cong denote the similarity of two REs. Two regular expressions r_1 and r_2 are similar, formally: $r_1 \cong r_2$, if they respect one of the following rules shown in table 11:

$(r) \& (s)$	\cong	$(s) \& (r)$
$(r) \& (\epsilon)$	\cong	r
$(r \& s) \& (t)$	\cong	$r \& s \& t$
$(\emptyset) \& (r)$	\cong	\emptyset
$(r \cdot s) \cdot t$	\cong	$r \cdot (s \cdot t)$
$\emptyset \cdot r$	\cong	\emptyset
$r \cdot \emptyset$	\cong	\emptyset
$\epsilon \cdot r$	\cong	r
$r \cdot \epsilon$	\cong	r
$(r s) t$	\cong	$r (s t)$
$\emptyset r$	\cong	r
$r \emptyset$	\cong	r
$(r^*)^*$	\cong	r^*
ϵ^*	\cong	ϵ

Table 6.2: Similarity rules

Note 6.4 Now we can rewrite the ACA, substituting the equality by similarity as follows: $\exists q_1 \in Q$ such that $q_1 \cong q_\alpha$.

Lemma 6.5 If two regular expressions are similar, then they are equivalent.

Example 6.6 Consider the REs $r_1 = \alpha \cdot b(c \cdot d)$ and $r_2 = (\alpha \cdot b \cdot c) \cdot d$. These REs are similar as is indicated in table 11 and equivalent because they describe the same regular language.

Example 6.7 Consider the RE $r = ab | cd$. We want to construct the automaton using derivatives.

We give each symbol and id. $r = a_1b_2 \mid c_1d_2$.
 $\text{First}(r) = \{\alpha, c\}$. $r_1 = D_\alpha(r) = b$, $r_2 = D_c(r) = d$.
 $\text{First}(r_1) = \{b\}$, $r_3 = D_b(r_1) = \epsilon$
 $\text{First}(r_2) = \{d\}$, $D_d(r_2)$

The DFA is shown in figure 2:

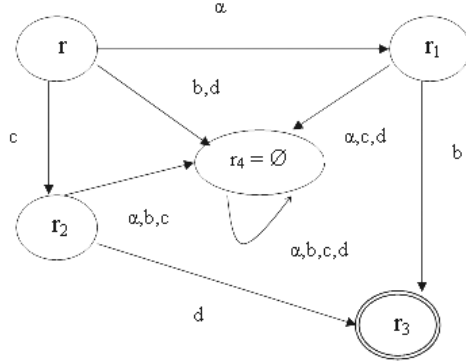


Figure 6.1: Finite state automaton

Example 6.8 Consider the regular expression: $r = abc^*$
 $(def^* \mid gh)xy$. We want to construct the automaton describing the \mathcal{L}_r .

We give an id to each symbol in the regular expression: $r = \alpha_1b_2c_3^*(d_3e_4f_5^* + g_3h_4)x_5y_6$

$\text{First}(r) = \{\alpha\}$, $r_1 = D_\alpha(r) = b_2c_3^*(d_3e_4f_5^* + g_3h_4)x_5y_6$

$\text{First}(r_1) = \{b\}$, $r_2 = D_b(r_1) = c_3^*(d_3e_4f_5^* + g_3h_4)x_5y_6$

$\text{First}(r_2) = \{c,d,g\}$, $D_c(r_2) = r_2$, $r_3 = D_d(r_2) = e_4f_5^*x_5y_6$, $r_4 = D_g(r_2) = h_4x_5y_6$

$\text{First}(r_3) = \{e\}$, $\text{First}(r_4) = \{h\}$, $r_5 = D_e(r_3) = f_5^*x_5y_6$, $r_6 = D_h(r_4) = x_5y_6$

$\text{First}(r_5) = \{f,x\}$, $\text{First}(r_6) = \{x\}$, $D_f(r_5) = r_5$, $r_7 = D_x(r_5) = y_6$, $D_h(r_6) = r_7$

$\text{First}(r_7) = \{y\}$, $r_8 = D_y(r_7) = \epsilon$

Note 6.9 Having all the derivatives of r , we have the states of the automaton. We can construct the automaton as shown in the previous example.

Example 6.10 Consider the Regular expression
 $r = ((ab) \& (cd))^*$. We want to calculate all the possible derivatives of r .

$$\begin{aligned}
r &= ((\alpha_1 b_2) \& (c_1 d_2))^* \\
\text{First}(r) &= \{\alpha, c\}, r_1 = D_\alpha(r) = b_2 \& (c_1 d_2) \cdot r, r_2 = D_c(r) = d_2 \& (\alpha_1 b_2) \cdot r \\
\text{First}(r_1) &= \{b, c\}, r_3 = D_b(r_1) = (c_1 d_2) \cdot r, r_4 = D_c(r_1) = (d_2 \& b_2) \cdot r \\
\text{First}(r_2) &= \{\alpha, d\}, D_\alpha(r_2) = b_2 \& d_2 = r_4, r_5 = D_d(r_2) = (\alpha_1 b_2) \cdot r \\
\text{First}(r_3) &= \{c\}, r_6 = D_c(r_3) = d_2 \cdot r \\
\text{First}(r_4) &= \{b, d\}, D_b(r_4) = r_6, r_7 = D_d(r_4) = b_2 \cdot r \\
\text{First}(r_5) &= \{a\}, D_\alpha(r_5) = r_7 \\
\text{First}(r_6) &= \{d\}, D_d(r_6) = r \\
\text{First}(r_7) &= \{b\} = r
\end{aligned}$$

Remark 6.11 *It is obvious from the example that during the derivation process we end up examining the same derivative several times.*

Chapter 7

DFA construction optimization

In this chapter, we provide an optimization for automata construction by avoiding the calculation of same derivatives more than once. The intuitive idea is based on the fact that the symbol with respect to which the derivation proceeds, is consumed during the derivation. So we can simulate the derivation process by giving each symbol in the regular expression an identifier helping to determine the position of the symbol.

Example 7.1 *Consider $r = xy$, we want to calculate all the possible derivatives of r .*

First we want give each symbol an id as follows:

$$\text{id}(x) = 1$$

$$\text{id}(y) = 2$$

The id of x is 1 because it is the initial of the regular expression and the id of y is 2 because it is concatenated x. We construct a derivative array putting the initial ids to the first line and subtracting one to every of the next lines until all ids are zero as follows:

	x	y
1.	1	2
2.	0	1
3.	0	0

The process simulates the derivation process where each symbol is consumed.

Each line of the array is then translated to a regular expression in a straightforward manner. We consider that the symbols with null ids are not occurring in the regular expression. If all the ids are zero, then the derivative is ϵ . The derivatives are shown below:

1. ab
2. a
3. ϵ

Example 7.2 Consider $r = x \mid yz$. We want to calculate all the possible derivatives.

We give the ids as follows:

$$\begin{aligned} \text{id}(x) &= 1 \\ \text{id}(y) &= 1 \\ \text{id}(z) &= 2 \end{aligned}$$

Ids of x and y are 1 because they are separated by alternation.

The derivative array is as follows:

	x	y	z
1.	1	1	2
2.	0	0	1
3.	0	0	0

The derivatives are:

1. $x \mid yz$
2. z
3. ϵ

Example 7.3 Consider $r = ab(c \mid de)fg$. We want to calculate all the possible derivatives.

We apply the derivative array method for each one of the following regular expressions: $r_1 = ab$, $r_2 = c \mid de$, $r_3 = fg$.

As a result:

For r_1 the derivatives are:

1. ab
2. b

3. ϵ

For r_2 the derivatives are:

1. $c \mid de$
2. e
3. ϵ

For r_3 the derivatives are:

1. fg
2. g
3. ϵ

Then we combine the above derivatives. We consider the derivatives of r_1 leaving the rest of r as it is and when all of the symbols of r_1 have been consumed we go on with the rest and so on. The result is the one follows:

1. $ab(c \mid de)fg$
2. $b(c \mid de)fg$
3. $(c \mid de)fg$
4. efg
5. fg
6. g
7. ϵ

Example 7.4 Consider $r = (ab(c \mid de)fg) \& (xy)$, we want to calculate all the possible derivatives of r .

We consider $r_1 = ab(c \mid de)fg$ and $r_2 = xy$. Due to the fact that r_1 and r_2 are separated by shuffle, we have to combine each derivative of r_1 with each derivative of r_2 . So we have 21 derivatives. The process of combining the derivatives of both subexpressions is quadratic. The possible derivatives are the one shown below:

We translate each line of the array to a RE. If the id of a symbol is zero then the symbol does not exist in the RE. The REs are shown below:

1. $(ab(c \mid de)fg) \& (xy)$
2. $(b(c \mid de)fg) \& (xy)$
3. $((c \mid de)fg) \& (xy)$
4. $(efg) \& (xy)$
5. $(fg) \& (xy)$
6. $(g) \& (xy)$
7. xy
8. $(ab(c \mid de)fg) \& (y)$
9. $(b(c \mid de)fg) \& (y)$
10. $((c \mid de)fg) \& (y)$
11. $(efg) \& (y)$
12. $(fg) \& (y)$
13. $(g) \& (y)$
14. y
15. $ab(c \mid de)fg$
16. $b(c \mid de)fg$
17. $(c \mid de)fg$
18. efg
19. fg
20. g
21. ϵ

The advantage of calculating the derivatives by using the derivative array is that we eliminate the calculation of same derivatives more than once. Specially in the case of regular expressions containing the shuffle operator the number of derivatives using the algorithm described in [7] is exponential to the cardinality of symbols of the regular expression.

Having the above derivatives we have all the states of the automaton. The transitions can easily be found. For two derivatives d_1 and d_2 we know that if: $|\text{Sym}(d_1)| - |\text{Sym}(d_2)| = 1$ there is a transition between d_1 and d_2 states, with transition symbol the symbol that is not contained in $\text{Sym}(r_1) \cap \text{Sym}(r_2)$.

<i>From</i>	<i>To</i>
1	2 and 8
2	3 and 9
3	4 and 10
4	5 and 11
5	6 and 12
6	7 and 13
7	14
8	9 and 15
9	10 and 16
10	11 and 17
11	12 and 18
12	13 and 19
13	14 and 20
14	21
15	16
16	17
17	18
18	19
19	20
20	21

Example 7.5 Consider $r = ab^*c^*de$. We want to calculate all the possible derivatives.

We give each symbol an id following the above rules: $\text{id}(a) = 1$, $\text{id}(b) = 2$, $\text{id}(c) = 3$, $\text{id}(d) = 3$, $\text{id}(e) = 4$.

We assign to d the same id as for c because d is going to be consumed when c is consumed. It is impossible that d appears in the first position of a derivative. For example consider a string $u = abcde$. $u \in \mathcal{L}_r$. Then if we derive wrt to $u_1 = abc$ we have: $D_{u_1}(r) = c^*de$, if we go on and derive wrt to $u_2 = abcd$ we have: $D_{u_2}(r) = e$.

We construct the derivative array for the regular expression as follows:

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
1.	1	2	3	3	4
2.	0	1	2	2	3
3.	0	0	1	1	2
4.	0	0	0	0	1
5.	0	0	0	0	0

We translate the above array into regular expression derivatives as follows:

1. ab^*c^*de
2. b^*c^*de
3. c^*de
4. e
5. ϵ

Example 7.6 Consider $r = (ab(c | de)fg)^*$. We want to calculate all the possible derivatives.

In this case we calculate the derivatives as in example 7.3 for the RE $r_1 = ab(c | de)fg$ but we consider that ϵ is not one of the derivatives of r . Due to the existence of kleene star $Sym(D_u(r)) \neq \emptyset, \forall u$.

Remark 7.7 It is easy the automata of r to be constructed using the automata of r_1 .

Note 7.8 The above algorithm, as it is described by the examples, provide a more efficient way to calculate the derivatives for a regular expression for automata construction. As we showed in the examples we avoid repetition of the calculation of derivatives more than once as described in [12]. Specially in the case of nested nullable regular expressions where the calculation of derivatives is exponential the above algorithm seems to be quadratic. Comparing it with the linear algorithm for membership checking described in [7], our algorithm seem to be applicable in a larger family of conflict free regular expressions.

Chapter 8

Conclusion and future work

The main aspect of our work is to provide a way to succeed XML on-line validation avoiding the automata construction. All the previous works based on automata construction provide the conditions under which the size of the automaton is not exponential. It is shown that only on a particular class of regular expressions, automata construction is an efficient way to succeed on-line validation.

Our approach is based on residual techniques. Particularly we use Brzozowski derivatives. The derivatives are considered an elegant and efficient way to construct recognizers succeeding the minimal states of automata. We used this technique on conflict free regular expression to provide on-line validation on the fly. We studied the membership checking of a string wrt to a regular expression, which is a simplification of XML on-line validation. We defined a set of rules extended with shuffle, taking advantage of the conflict free property of the regular expression. We avoided a considerable number of derivations compared with the rules provided by Brzozowski. We calculated the time complexity for membership checking without automata construction, by applying the rules we have defined on conflict free REs.

Furthermore, we provided a way to use our rules on automata construction eliminating a considerable number of calculation. Nevertheless, we have not succeeded the optimal calculation process. At the last part we provide in the form of examples an algorithm that provide an optimization on the number of calculations for automata construction. We believe that this algorithm is applicable to every class of conflict free regular expression.

Our future work has two directions: Firstly we want to examine more in depth the optimized algorithm for automata construction, giving the potential classes of conflict free regular expressions that it is applicable. Further-

more, we want to provide the time complexity of this algorithm and compare it with the already known ones for automata construction. Secondly, we want to study more the on-line validation technique avoiding automata construction. We want to provide optimization on different classes of regular expressions, theoretically, and examine the efficiency of this technique in real worlds DTDs.

Bibliography

- [1] B. K. Anne and D. Wood. One unambiguous regular languages. *Information and computation*, 1998.
- [2] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise dtDs from xml data. *VLDB*, 2006.
- [3] G. J. Bex, F. Neven, and S. Vansummeren. Inferring xml schema definitions from xml data. *VLDB*, 2007.
- [4] J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 1964.
- [5] C. Chitic and D. Rosu. On validation of xml streams using finite state machines. *WebDB*, 2004.
- [6] W. Fan and L. Libkin. On xml integrity constraints in the presence of dtDs. *ACM*, 2002.
- [7] G. Ghelli, D. Colazzo, and C. Sartiani. Linear time membership for a class of xml types with interleaving and counting. *PLAN-X*, 2008.
- [8] A. H. Laender, M. M. Moro, C. Nascimento, and P. Martins. An x-ray on web-available xml schemas. *SIGMOD*, 2009.
- [9] C. S. McQueen. Application of brzozowski derivatives to xml schema processing. *Extreme Markup Languages*, 2005.
- [10] M. Montazerian, P. T. Wood, and S. R. Mousavi. Xpath query satisfiability is in ptime for real-world dtDs. *Xsym*, 2007.
- [11] M. Murata, D. Lee, M. Mani, and K. Kawagushi. Taxonomy of xml schema languages using formal language theory. *ACM*, 2005.
- [12] S. Owens, J. Reppy, and A. Turon. Regular expression derivatives reexamined. *Appel*, 1988.

- [13] R. Price. Beyond sgml. *ACM*, 1998.
- [14] L. Segoufin and C. Sirangelo. Constant-memory validation of streaming xml documents against dtds. *ICDT*, 2007.
- [15] L. Segoufin and V. Vianu. Validating streaming xml documents. *ACM PODS*, 2002.
- [16] P. T. Wood. Containment for xpath fragments under dtd constraints. *ICDT*, 2003.