XML and Databases

Lecture 5: XML Validation using Automata

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

Outline

1. Recap: deterministic Reg Expr’s / Glushkov Automaton
2. Complexity of DTD validation
3. Beyond DTDs: XML Schema and RELAX NG
4. Static Methods, based on Tree Automata

Outline

1. Recap: deterministic Reg Expr’s / Glushkov Automaton
2. Complexity of DTD validation
3. Beyond DTDs: XML Schema and RELAX NG
4. Static Methods, based on Tree Automata

Previous Lecture

XML type definition languages

want to specify a certain subset of XML doc’s = a “type” of XML documents

Remember

The specification/type definition should be simple, so that

→ a validator can be built automatically (and efficiently)
→ the validator runs efficient on any XML input

(similar demands as for a parser)

→ Type def. language must be SIMPLE!

(similarity: parser generators use EBNF or smaller subclasses: LL /LR)

O(n^3) parsing

XML Type Definition Languages

DTD (Document Type Definition, W3C)

Originate from SGML. Now part of XML

→ DTD may appear at the beginning of an XML document

XML Schema (W3C)

Now at version 1.1

HUGE language, many built-in simple types

→ Schemas themselves: written in XML

See the “Schema Primer” at http://www.w3.org/TR/xmlschema-0/

RELAX NG (Oasis)

For tree structure definition, more powerful than Schemas & DTDs

XML Type Definition Languages

DTD (Document Type Definition)

<!DOCTYPE root-element [ doctype declaration …]>

<!ELEMENT element-name content-model>

content-models

• EMPTY
• ANY
• #PCDATA | element-name_1 | … | element-name_n
• deterministic Reg Expr over element names

<!ATTLIST element-name attr-name attr-type attr-default …>

Types: CDATA, (v1|…), ID, IDREFs

Defaults: #REQUIRED, #IMPLIED, “value”, #FIXED

Most interesting / challenging aspect of DTDs

XML Type Definition Languages

DTD (Document Type Definition)

<!DOCTYPE root-element [ doctype declaration …]>

<!ELEMENT element-name content-model>

content-models

• EMPTY
• ANY
• #PCDATA | element-name_1 | … | element-name_n
• deterministic Reg Expr over element names

<!ATTLIST element-name attr-name attr-type attr-default …>

Types: CDATA, (v1|…), ID, IDREFs

Defaults: #REQUIRED, #IMPLIED, “value”, #FIXED
Summary

In order to check whether a (large) document is valid with respect to a given DTD ("it validates") you need to:
- check if children lists match the given regular expressions.
This can be done efficiently, using finite-automata (FAs).

To check if a regular expression \( e \) is allowed in a DTD we have to construct a particular finite automaton: the Glushkov automaton. \( \text{Glu}(e) \) must be deterministic.

Note: If \( \text{Glu}(e) \) is deterministic, then its size (# transitions) is linear in size(e)!

Question: Can you explain why this is the case?

More Notes

1. From a deterministic FA you cannot necessarily obtain a deterministic (= 1-unambiguous) regular expression!
   Example: \( e = (a | b)^* a (a | b) \) \( \Leftarrow \) NO 1-unambiguous reg exp exists for \( e \)

2. \( \text{Glu}(e) \) is closely related to \( \text{Thomson}(e) \) \text{[remove \( \varepsilon \)-transitions]} and to \( \text{Berry/Seth}(e) \) \text{[same]} and \( \text{Brzozowski}(e) \)

Question: Can you explain why this is the case? for more details:
See paper by Brüggemann-Klein, linked from the course web-page.

Glushkov automaton \( \text{Glu}(e) \)

Each letter-position in the regular expression becomes one state of \( \text{Glu} \); plus, \( \text{Glu} \) has one extra begin state.

\[ \text{FIRST}(e) = \{ \text{all possible begin positions of words matching } e \} \]

\[ \text{FIRST}(R (E | G) (EX)^*) = \{ \text{RI} \} \]
Glushkov’s automaton

Character in RE = state in automaton
+ one state for the beginning of the RE

Transitions show which characters/positions can precede each other

\[ R( \text{E} | \text{G} )( \text{EX} )^* \]

Glushkov automaton $G(e)$

- Each position in the Reg Expr $e$ becomes one state of $G$; plus, $G$ has one extra begin state.
- $\text{FIRST}(e)$ = all possible begin positions of words matching $e$
  
  \[ \text{FIRST}(R (E | G)(EX)^*) = \{ R_1 \} \]

- $\text{FOLLOW}(e, x)$ = all possible positions following position $x$ in $e$
  
  \[ \text{FOLLOW}(R (E | G)(EX)^*, R_1) = \{ E_2, G_3 \} \]

  - From state "$R_1$": add $E$-transition to $E_2$
    
    $G$-transition to $G_3$

$\text{Glushkov's automaton}$

- Character in $RE = \text{state}$ in automaton
  + one state for the beginning of the RE

- Transitions show which characters/positions can precede each other

\[ R (E | G)(EX)^* \]

- $\text{FOLLOW}(e, R_1)$ =
- $\text{FOLLOW}(e, G_3)$ =
- $\text{FOLLOW}(e, E_4)$ =

$\text{Glushkov's automaton}$

- Character in $RE = \text{state}$ in automaton
  + one state for the beginning of the RE

- Transitions show which characters/positions can precede each other

\[ R (E | G)(EX)^* \]

- $\text{FOLLOW}(e, E_j) =$

$\text{Glushkov's automaton}$

- Character in $RE = \text{state}$ in automaton
  + one state for the beginning of the RE

- Transitions show which characters/positions can precede each other

\[ R (E | G)(EX)^* \]

- $\text{FOLLOW}(e, E_j) =$

- $\text{FOLLOW}(e, G_j) =$

- $\text{FOLLOW}(e, R_j) =$
Glushkov’s automaton
- Character in RE = state in automaton
  + one state for the beginning of the RE
- Transitions show which characters/positions can precede each other

\[
R \ (E \mid G) \ (E \mid X)^* \]

FOLLOW( e, X5) =  

- Each position in the Reg Expr e becomes one state of G; plus, G has one extra begin state.
- \( \text{FIRST}(e) \) = all possible begin positions of words matching e
  - e.g. \( \text{FIRST}(R \ (E \mid G) \ (E \mid X)^*) = \{R_1\} \)
- \( \text{FOLLOW}(e, x) \) = all possible positions following position x in e
  - e.g. \( \text{FOLLOW}(R \ (E \mid G) \ (E \mid X)^*, R_1) = \{E_2, G_3\} \)
  
  - From state “R1”: add E-transition to E2
    - G-transition to G3
- \( \text{LAST}(e) \) = all possible end positions of words matching e
  - e.g. \( \text{LAST}(R \ (E \mid G) \ (E \mid X)^*) = \{E_2, G_2, X_5\} \)

Is this automaton deterministic??
Glushkov automaton $G(e)$
Each position in the Reg Expr $e$ becomes one state of $G$; plus, $G$ has one extra begin state.

$\text{FIRST}(e) =$ all possible begin positions of words matching $e$

$\text{FOLLOW}(e, x) =$ all possible positions following position $x$ in $e$

$\text{LAST}(e) =$ all possible end positions of words matching $e$

Naïve implementation: $O(n^3)$ time, where $n =$ size($e$)
(for each position: computing FOLLOW goes through every position at each step, needs to compute union $\cup O(n^3)$)

Not really needed. Can be improved to $O(n^2)$

Can be improved to $O(\text{size}(e) + \text{size}(G(e)))$

Naïve implementation: $O(n^3)$ time, where $n =$ size($e$)
(for each position: computing FOLLOW goes through every position at each step, needs to compute union $\cup O(n^3)$)

Not really needed. Can be improved to $O(n^2)$

Can be improved to $O(\text{size}(e) + \text{size}(G(e)))$

To avoid these expensive running times

$\text{DTD}$ requires that $F_A + G(e)$ must be deterministic!

$n =$ length($e$)  
$m =$ size($e$)

If $s =$ number($e$) is assumed fixed (not part of the input)

Total Running time $O(n + m)$

Otherwise: $O(n + ms)$

How can you implement a regular expression?

Input: Reg Expr $e$, string $w$
Question: Does $w$ match $e$?

Algorithm:
$\text{FA} = \text{BuildFA}(e)$
$\text{DFA} = \text{BuildDFA}(\text{FA})$

Size of FA is linear in size($e$),$m$
Size of DFA is exponential in $m$

Total Running time $O(s^2)$

Other alternative: $O(nm)$

Unrestricted  Reg Expr $e$

Now that we know how the check all the different content-models (in particular det. Reg Expr’s) how to build full validator for a DTD?

elem-name$_1$  $\rightarrow$ RegExpr$_1$

elem-name$_2$  $\rightarrow$ RegExpr$_2$

...  $\rightarrow$  Automata A$_1$, A$_2$, ..., A$_k$

elem-name$_k$  $\rightarrow$ RegExpr$_k$

The Validation Problem
Given a DTD $T$ and a document $D$, is $D$ valid wrt $T$?

Top-Down Implementation
$\rightarrow$ at element node $w$, label elem-name$_{i,j}$, run automaton A$_i$

$\rightarrow$ check attribute constraints

$\rightarrow$ check ID/IDREF constraints

Total Running time
linear in the sum of sizes of the DTD and the document. $O(\text{size}(T) + \text{size}(D))$

Summary
Deterministic (1-unambiguous) content models give rise to efficient matching algorithms.

(they avoid $O(nm)$ or $O(n^2m^2)$ complexities)

Disadvantages
$\rightarrow$ Hard to know whether given reg expr is OK (deterministic)

$\rightarrow$ Det. reg exprs are NOT closed under union. (not so nice...)

Question Can you see why?

Hint: find det. reg. exprs $e_1$ and $e_2$ such that their union is equal to $(a | b)^*$ # $(a | b)$
DTDs have the "label-guarded subtree exchange" property:

$t_1, t_2$ trees in a DTD language $T$
$v_1$ node in $t_1$, labeled "lab"
$v_2$ node in $t_2$, labeled "lab"

Trees obtained by exchanging the subtrees rooted at $v_1$ and $v_2$ are also in $T$.

aka "local"

Beyond DTDs

Often, the expressive power of DTDs is not sufficient.

Problem: each element name has precisely one content-model in a DTD.

Would like to distinguish, depending on the context (parent).

"specialization"

"content model only depends on label of parent"

Beyond DTDs

Often, the expressive power of DTDs is not sufficient.

Problem: each element name has precisely one content-model in a DTD.

Would like to distinguish, depending on the context (parent).

car has different structure, in different contexts.

Specialized DTDs

Dealer $\rightarrow$ dealer [Used, New]
Used $\rightarrow$ used [CarUsed]*
New $\rightarrow$ new [CarNew]*
CarUsed $\rightarrow$ car [Model, Year]
CarNew $\rightarrow$ car [Model]

New notation. Use capitalized TYPE Names

A grammar $G$ is local, if for any $\text{label}[\text{RegExpr}_1], \text{label}[\text{RegExpr}_2]$ present in $G$

it holds that $\text{RegExpr}_1 = \text{RegExpr}_2$.

By definition: Every DTD is a local grammar, and vice versa.
A grammar $G$ is *local*, if for any \( \text{label}[\text{RegExp}_1], \text{label}[\text{RegExp}_2] \) present in $G$ it holds that $\text{RegExp}_1 = \text{RegExp}_2$.

By definition: Every DTD is a *local* grammar, and vice versa.

A grammar $G$ is *single-type*, if for any \( \text{label}[\text{RegExp}_1], \text{label}[\text{RegExp}_2] \) occurring in the same rule of $G$ it holds that $\text{RegExp}_1 = \text{RegExp}_2$.

Classes of Grammars
- **local**: no competing TYPE names! (DTDs)
- **single-type**: TYPE names in the same content model do not compete! (XML Schemas)
- **regular**: no restriction… (RELAX NG)

Through the use of TYPE Names (nonterminals / states) you can distinguish deep context!

Can we model context that is far away from the specialized node?
New notation. Use capitalized TYPE Names

Can we model context that is far away from the specialized node? Sure!

Question Is this grammar single-type?

prev. example: probably, not expressable in single-type (XML Schema).

Other example:

<table>
<thead>
<tr>
<th>Person</th>
<th>MPerson</th>
<th>FPerson</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPerson</td>
<td>FPerson</td>
<td>Person</td>
</tr>
<tr>
<td>Person</td>
<td>MPerson</td>
<td>FPerson</td>
</tr>
<tr>
<td>Person</td>
<td>Male</td>
<td>Female</td>
</tr>
<tr>
<td>Person</td>
<td>FSpouse</td>
<td>MSpouse</td>
</tr>
<tr>
<td>Person</td>
<td>Children</td>
<td>Person</td>
</tr>
</tbody>
</table>

a person's spouse must have opposite gender.

Note This example and the Pet-example are taken from Hosoya's book (see course web page).
Classes XML Type Formalisms

- **local**: no competing TYPE names! (DTDs)
- **single-type**: TYPE names in the same content model do not compete! (XML Schema’s)
- **regular**: no restriction… (RELAX NG)

**Increasing Expressiveness**

**Questions**

Given two DTDs D1 and D2 can we check if
- all documents valid for D1 are also valid for D2? (DTD inclusion problem)
- D1 and D2 describe the same set of documents? (DTD equality problem)

Given a Relax NG grammar G, can we check if
- there exists any document that is valid for G? (emptiness problem)
- there is a document valid for G and valid for G2? (intersection & emptiness)

All of the checks can be done automatically, for regular tree grammars!

**Tree Automata**: very powerful framework,
- Have all the good properties of string automata!
- Yet, they are more expressive!

**Note**

String automata are not sufficient to check DTDs / Schemas!
Even if we only consider well-bracketed strings!

**Example 1**

- \( c \rightarrow [a,c,b] \)
- \( a \rightarrow [a,c,a] \)
- \( b \rightarrow \) empty
- \( c \rightarrow \) empty

**Example 2**

- \( a \rightarrow [a,a,b] \)
- \( a \rightarrow [a,a,b] \)
- \( b \rightarrow \) empty
- \( a / b / c \rightarrow \) empty

Finite-state automata are important:
- Think you are in a maze, with only fixed memory and you can only read the maze (cannot mark anything).

Model by finite automaton. In state \( q1 \),
\( q2 \rightarrow [N|S|E|W] \)
\( q3 \rightarrow [N|S|E|W] \)
\( q4 \rightarrow [N|S|E|W] \)

Can an automation search the maze?
Can an automaton search the maze?
No!!
Æ need markers (“pebbles”).
How many? 5? 2?

All of the checks can be done automatically, for regular tree grammars!
constant memory
computation

Finite-state automata are important:
Æ Think you are in a maze, with only fixed memory and you can only read the maze (cannot mark anything).
Model by finite automaton. In state q1, (to [N|S|E|W], wall ) ➔ ( q2, [N|S|E|W] )
qu1
Can an automaton search the maze?
No!! ➔ need markers (“pebbles”). How many? 5? 2?

4. Static Methods, based on Tree Automata

Given grammars D1 and D2 can we check if
Æ all documents valid for D1 are also valid for D2? (inclusion problem)
Æ D1 and D2 describe the same set of documents? (equality problem)
Æ does there exists any document that is valid for D1? (emptiness problem)
Æ there is a document valid for D1 "and" valid for D2? (intersection & emptiness)

Æ The checks above give rise to very powerful optimization procedures for XML Databases!

For example: documents d_1, d_2, ..., d_n are valid for your schema “Small_xhtml”.
Are they also valid for schema XHTML?
Æ Check inclusion problem for Small_html and XHTML!

4. Static Methods, based on Tree Automata

Person ➔ MPerson | FPerson
MPerson ➔ person [Name, gender[Male], FSpouse?]
FPerson ➔ person [Name, gender[Female], MSpouse?]

Regular Tree Grammar
Rules of the form
Leaves may be labeled
by TypeNames
TypeName ➔ Tree
person
Name gender FSpouse
Male

Alternatively, regular tree languages are defined by Tree Automata.
state, element-name ➔ state1, state2
conventionally, defined for binary/ranked trees.

Automata on words

4. Static Methods, based on Tree Automata

Given grammars D1 and D2 can we check if
Æ all documents valid for D1 are also valid for D2? (inclusion problem)
Æ does there exists any document that is valid for D1? (emptiness problem)
Æ there is a document valid for D1 "and" valid for D2? (intersection & emptiness)

Æ The checks above give rise to very powerful optimization procedures for XML Databases!

For example: documents d_1, d_2, ..., d_n are valid for your schema “Small_xhtml”.
Are they also valid for schema XHTML?
Æ Check inclusion problem for Small_html and XHTML!

4. Static Methods, based on Tree Automata

Given grammars D1 and D2 can we check if
Æ all documents valid for D1 are also valid for D2? (inclusion problem)
Æ D1 and D2 describe the same set of documents? (equality problem)
Æ does there exists any document that is valid for D1? (emptiness problem)
Æ there is a document valid for D1 "and" valid for D2? (intersection & emptiness)

Æ The checks above give rise to very powerful optimization procedures for XML Databases!

For example: documents d_1, d_2, ..., d_n are valid for your schema “Small_xhtml”.
Are they also valid for schema XHTML?
Æ Check inclusion problem for Small_html and XHTML!
Automata on words

Expressiveness
- deterministic ≠ nondeterministic
- left-to-right ≠ right-to-left

word is accepted by the automaton

Automata on trees

1. bottom-up LABEL( state1, state2 ) \rightarrow state

\begin{align*}
0() & \rightarrow F \\
1() & \rightarrow T \\
\text{OR}(F,F) & \rightarrow F \\
\text{OR}(F,T) & \rightarrow T
\end{align*}
Automata on words

Expressiveness:
- Deterministic ≠ Nondeterministic
- Left-to-right ≠ Right-to-left

Exp. blow-up = more succinct on words

Automata on trees

1. Bottom-up

\[ \text{LABEL}(\text{state}_1, \text{state}_2) \rightarrow \text{state} \]

\[
\begin{align*}
\{0\} & \rightarrow F \\
\{1\} & \rightarrow T \\
\{0, F\} & \rightarrow F \\
\{0, T\} & \rightarrow T \\
\{1, F, T\} & \rightarrow F \\
\end{align*}
\]

Tree is accepted by the automaton

Question:

How much memory do you need exactly, to run such a bottom-up tree automaton?

Similarly as for word automata:

For every nondeterministic bottom-up tree automaton, there is an equivalent deterministic bottom-up tree automaton.

Again, the construction can cause exponential size blow-up.

2. Top-down

\[ \text{state}, \text{LABEL} \rightarrow (\text{state}_1, \text{state}_2) \]

\[
\begin{align*}
\text{begin}, a & \rightarrow (\text{any}, \text{find}_a) \\
\text{begin}, b & \rightarrow (\text{begin}, \text{any}) \\
\text{find}_a, a & \rightarrow (\text{find}_a, \text{any}) \\
\text{find}_a, b & \rightarrow \text{ACC} \\
\text{find}_b, e & \rightarrow \text{REJ} \\
\text{any}, a/b & \rightarrow (\text{any}, \text{any}) \\
\text{any}, \text{b/e} & \rightarrow \text{ACC}
\end{align*}
\]

Tree is accepted by the automaton

Nondeterministic:

Tree must have a leftmost path with a $-node.

"Top-most a-node on the left-most path must have a right subtree which contains a $-node."
For every nondeterministic bottom-up tree automaton there is an equivalent deterministic bottom-up tree automaton.

**Question**
Can you find an equivalent bottom-up automaton for this example?

1. **top-down** state, LABEL \(\rightarrow\) (state1, state2)

   - must contain a $-leaf
   - a,b = binary node labels
   - e,$ = leaf node labels
   - "top-most a-node on the left-most path must have a right-subtree which contains a $-node."

   - begin, a \(\rightarrow\) (any, find$)
   - begin, b \(\rightarrow\) (begin, any)
   - find$, a/b \(\rightarrow\) { (find$, any), (any, find$) }
   - find$, e \(\rightarrow\) ACC
   - find$, $ \(\rightarrow\) REJ
   - any, a/b \(\rightarrow\) (any,any)
   - any, $/e \(\rightarrow\) ACC
   - nondeterministic

   - accept tree if there exists an accepting run

**Question**
Is there an equivalent deterministic top-down automaton??

- NO! ☻

This set of two trees canNOT be recognized by any deterministic top-down tree automaton!!

**Why?**

Questions

What about **local** tree languages (defined by DTDs)?

- Can they be accepted by deterministic top-down automata?

What about **single-type** tree languages (defined by XML Schema’s)?

- Can they be accepted by deterministic top-down automata?

Yes!

Hence, there is no DTD / Schema for \{ name[first, last], name[last, first] \}
For every deterministic bottom-up tree automaton there exists a minimal unique equivalent one!

Equivalence is decidable

In fact, YOU have already produced minimal bottom-up tree automata!

The minimal DAG of a tree $t$ can be seen as the minimal unique tree automaton that only accepts the tree $t$.

**Question**

How expensive (complexity) to find minimal one?

Same as for word automata?

---

Tree Automata are a very useful concept in CS!

- Heavily used in verification

  "Derive a property of a complex object from the properties of its constituents..."

- Do all graphs / chip-layouts produced in this way, have property P?

  Use the Hierarchical construction history of an object, in order to work on a "parses" tree instead of a complex graph.

  From there, use tree automata. 😊

  Many NP-complete graph problems become tractable on "bounded-treewidth" graphs!

---

XML

Tree Automata play crucial role for

- Efficient validators against XML Types

- Optimizations if doc1 is of TYPE1, then no need to validate against TYPE2, if we know TYPE2 included in TYPE1
  - if only "slightly different" then only need to validate "there"
  - incremental validation against updates
  - etc, etc.

- Efficient query evaluators, use richer automata which can select nodes and produce query answers

- Optimizations if answer of QUERY1 is in cache, then no need to evaluate QUERY2, if "included" in QUERY1.
  - if every possible answer set to QUERY1 (of TYPE X) is EMPTY, then no need to evaluate on the real data!

- XML Type Checking for Programming Languages

---

The Future

In 5-10 years from now:

- You can write a function in Programming Language X

  ```
  function foo(XMl document D: TYPE1): TYPE2 {
    traverse D & compute output;
    return output;
  }
  ```

  Compiler (XML Type Checker) will complain, if your function does not compute documents of TYPE2.

  - If no complaint, then **guaranteed**: ALL outputs are ALWAYS of correct type!!

- Compilers will have to be able to give static guarantees about input/output behaviour of program!
End
Lecture 5