

XML typing

MPRI 2.26.2: Web Data Management

Antoine Amarilli^a

Friday, December 14th



^aBased on slides from the Webdam book by Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart
<https://webdam.inria.fr/Jorge/files/TypingXML.ppt>

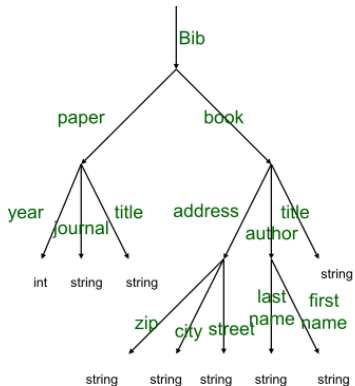
Introduction

XML typing

- XML documents must be **well-formed**
- However, **typing** is not required
- Why typing?
 - **Formal specification** of a format
 - Ensure **interoperability** between programs
 - Automatically detect **bugs** when generating
 - Make it easier to **author** a document
 - Make it easier to **query** it
 - Avoid introducing errors when **updating**
 - Make querying more **efficient**
- **Many languages!**

https://en.wikipedia.org/wiki/List_of_types_of_XML_schemas

Improving storage



```
select X.title
from Bib._X
where X.*.zip = "12345"
```



```
select X.title
from Bib.book X
where X.address.zip = "12345"
```

Verification

- **Easy:** when data is produced/received, check the schema at runtime
- **Hard:** verify statically that a program will **always** produce valid output
- Explored in the **Cduce** language <http://www.cduce.org/>
- Applicable to **programs**, to **queries** in XQuery, to **XSLT**, etc.

Verification example

```
for $p in doc("parts.xml")//part[color="red"]
return <part>
<name>{$p/name/text()}</name>
<desc>{$p/desc/node()}</desc>
</part>
```

- The type of the result will be:
(part (name (string) desc (any))*)
- If the type of parts.xml//part/desc is string
then the type will be:
(part (name (string) desc (string))*)

Generating complicated types

```
for $x in $d/*, $y in $d/*  
return <b/>
```

- When we run this on an input document of the form $(\langle a/\rangle)^n$ it will produce $(\langle b/\rangle)^{n^2}$
- No hope to describe this as a reasonable type

DTDs

Type declaration

- The simplest (and oldest) typing mechanism is based on *Document Type Definitions* (DTD).
- A DTD may be specified in the prologue with the keyword **DOCTYPE** using an ad hoc syntax.
- Note: the DTD is not itself an **XML document**

Example of an XML document with its DTD

```
<?xml version="1.0" standalone="yes" ?>
<!DOCTYPE email [
  <!ELEMENT email ( header, body )>
  <!ELEMENT header ( from, to, cc? )>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT from (#PCDATA)>
  <!ELEMENT cc (#PCDATA)>
  <!ELEMENT body (paragraph*) >
  <!ELEMENT paragraph (#PCDATA)> ]>
<email>
  <header>
    <from> af@abc.com </from>
    <to> zd@ugh.com </to>
  </header>
  <body></body>
</email>
```

Document Type Definition

A DTD may also be specified **externally** using an URI:

```
<!DOCTYPE docname SYSTEM "DTD-URI" [local-declarations]>
```

Some DTDs have established **public identifiers**:

```
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE html PUBLIC  
    "-//W3C//DTD XHTML 1.0 Transitional//EN"  
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Simple declarations in a DTD

Declaring an **empty element**:

```
<!ELEMENT br EMPTY>
```

Declaring an element containing **text**:

```
<!ELEMENT from (#PCDATA)>
```

Declaring an element containing **anything**:

```
<!ELEMENT from ANY>
```

Declaring an element containing **text or some elements** among a list:

```
<!ELEMENT from (#PCDATA | a1 | a2 | a3)*>
```

Note: there can be **only one** declaration per element name!

More complex declarations in a DTD

Declaring an element with complex content:

```
<!ELEMENT complex (E)>
```

where E is a **deterministic** regular expression built with:

- ,: concatenation
- |: or
- ?: 0 or 1 occurrence
- *: arbitrary number of occurrences
- +: non-zero number of occurrences

Whitespace that shouldn't exist is ignored

Deterministic regexp

- To make validation easier, the regexp must be **deterministic**: while matching any word of element names, there must never be any choice in how to continue
- **More formally**: for any position in the regexp and element name, there must be at most one position where we can go
- **Non-deterministic examples**: $(e1\ e2\ | \ e1\ e3)$ or $e1? \ e1$

Deterministic regexp

- To make validation easier, the regexp must be **deterministic**: while matching any word of element names, there must never be any choice in how to continue
- **More formally**: for any position in the regexp and element name, there must be at most one position where we can go
- **Non-deterministic examples**: $(e1\ e2\ | \ e1\ e3)$ or $e1? \ e1$
- Some regular languages have **no deterministic regular expressions**, e.g., $(a|b)^* \ a \ (a|b)$, see Bruggemann-Klein and Wood, *One-Unambiguous Regular Languages*, I&C 1998.
<https://core.ac.uk/download/pdf/82738828.pdf>
- We can test in **linear time** if a regular expression is deterministic: see Groz, Maneth, and Staworko, *Deterministic Regular Expressions in Linear Time*, PODS 2012.
<https://hal.inria.fr/inria-00618451/document>

Attributes

Declaring an attribute `att` on element `elt`:

```
<!ATTLIST element attribute type default>
```

The `type` can be one of the following (non-exhaustive):

- CDATA: `anything`
- `v1 | v2 | ... | vn`: one of these `values` (XML tokens)
- ID: a (unique) XML ID (at most one such attribute per element)
- IDREF: a `reference` to some ID value

The `default` can be:

- `"value"` Some `default value` (so the attribute is optional)
- `#IMPLIED` saying that the attribute is `optional` (no default)
- `#REQUIRED` saying that the attribute is `required`
- `#FIXED "value"`

Entities

DTDs allow us to define **entities** (like macros):

```
<!ENTITY author "John Doe">
```

and then to refer to them in the XML:

```
&author;
```

There are also mechanisms for **external entities**.

External DTDs

The use of external DTDs poses some **problems**:

- We may be **offline**
- They may be **down**
- Fetching them causes **loss of privacy**

If we do not want to access the external DTD, we:

- Cannot **validate** the document (duh)
- Cannot **replace** entities defined in the DTD
- Cannot **know** about default attribute values

XML Schema

Key points of XML Schema (XSD for XML Schema Definition)

- More **recent** and more **expressive** than the DTD standard
- More **complex**
- Support for **namespaces**
- Constraints on **data types**, e.g., dates, numbers, etc. (19 primitive types)
- May have multiple **definitions** for the same element in different contexts
- More powerful **uniqueness** constraints (can be scoped to part of the document)
- An XML schema is an **XML document** so there is an **XML Schema schema**:
`https://www.w3.org/2009/XMLSchema/XMLSchema.xsd`
- In XSD 1.1, **assertions** with XPath expressions

Example of XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="character" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="friend-of" type="xsd:string"
              minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="since" type="xsd:date"/>
            <xsd:element name="qualification" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="isbn" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

XML Schema simple types

- Built-in types: string, integer, date, time, ...
- Modifying basic types, e.g.:

```
<element name="age">  
  <simpleType>  
    <restriction base="integer">  
      <minInclusive value="0"/>  
      <maxInclusive value="150"/>  
    </restriction>  
  </simpleType>  
</element>
```

Also: enumeration of values, patterns, length, etc.

→ These simple types are used for attributes and for elements that contain only text

More XML Schema features

- `all`: we must have all elements but in an arbitrary order

```
<element name="person">
  <complexType>
    <all>
      <element name="firstname" type="string"/>
      <element name="lastname" type="string"/>
    </all>
  </complexType>
</element>
```

- Giving `names` to types and reusing them afterwards:

```
<complexType name="mytype">
  ...
</complexType>
<element name="myelt" type="mytype" />
```

- Assertions using XPath

More XML Schema features (2)

- **Groups** of elements and attributes
- **any** and **anyAttribute** to allow anything
- **Substitution groups** to indicate that an element can be substituted for another
- Including an **existing schema** (possibly in a different namespace)

```
<include schemaLocation="http://..." />  
<import namespace="http://..."  
        schemaLocation="http://..." />
```

- Redefining an existing schema:

```
<redefine schemaLocation="http://...">  
    ...  
</redefine>
```


XML Schema vs DTD

```
<!ELEMENT note
  (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"/>
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Criticism of XML Schema

- Very **complex**: 461 pages
- Extremely **verbose**
- Some unexpected **shortcomings**: no simple way to say what the root element must be
- Not sufficiently **expressive**, in particular not **self-describing**
- Still requires **deterministic** regular expressions

Beyond XML Schema

- Relax NG:
 - Simpler than XML Schema
 - More minimalistic default types
 - No way to specify a document's schema within the document
 - No way to add default values
 - No determinism requirement
- Schematron:
 - Different approach: list of rules that the document must pass
 - Can be used in conjunction with DTD/XSD/RelaxNG
- Many possible choices!
https://en.wikipedia.org/wiki/XML_schema#Languages

RELAX NG example

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="book">
      <oneOrMore>
        <ref name="page"/>
      </oneOrMore>
    </element>
  </start>
  <define name="page">
    <element name="page">
      <text/>
    </element>
  </define>
</grammar>
```

Also a short syntax:

```
start = element book { page+ }
page = element page { text }
```

Schematron example

```
<?xml version="1.0" encoding="UTF-8"?>
<sch:schema xmlns:sch="http://www.ascc.net/xml/schematron">
<sch:pattern name="Check structure">
<sch:rule context="Person">
  <sch:assert test="@Title">Person must have Title attribute</sch:assert>
  <sch:assert test="count(*)=2 and count(Name)=1 and count(Gender)=1">
    The element Person should have the child elements Name and Gender.
  </sch:assert>
  <sch:assert test="*[1] = Name">Name must be first.</sch:assert>
</sch:rule>
</sch:pattern>
<sch:pattern name="Check co-occurrence constraints">
<sch:rule context="Person">
  <sch:assert test="(@Title='Mr' and Gender='Male') or @Title!='Mr'">
    If the Title is "Mr" then the gender of the person must be "Male".
  </sch:assert>
</sch:rule>
</sch:pattern>
</sch:schema>
```

Tree automata

General presentation

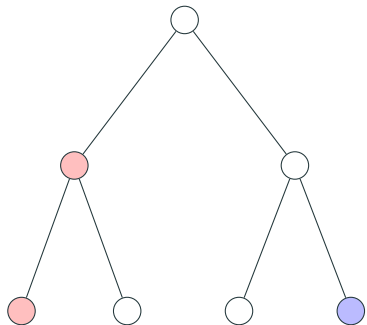
- Tree automata **generalize** word automata
- Remember that a deterministic **word automaton** A specifies:
 - An **alphabet** Σ of labels
 - A set Q of **states**
 - An **initial state** $q_0 \in Q$
 - A set $F \subseteq Q$ of **final states**
 - A **transition function** $\delta : Q \times \Sigma \rightarrow Q$

Automaton reminders

- The automaton can be:
 - **deterministic**: the transition function δ is a function
 - **non-deterministic**: δ is a relation $\subseteq Q \times \Sigma \times Q$;
we may also have ϵ -transitions
- We can **determinize** automata but at the cost of an **exponential blowup**
- Some languages, e.g., $\{a^n b^n \mid n \in \mathbb{N}\}$, cannot be recognized by a finite word automaton

Bottom-up tree automata

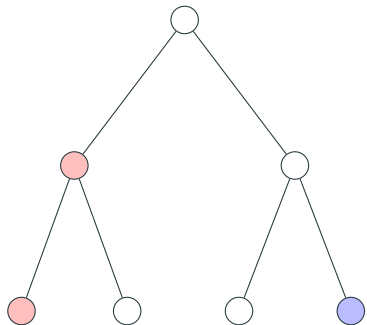
Alphabet Σ : ○ ● ●



Bottom-up tree automata

Alphabet Σ : ○ ● ●

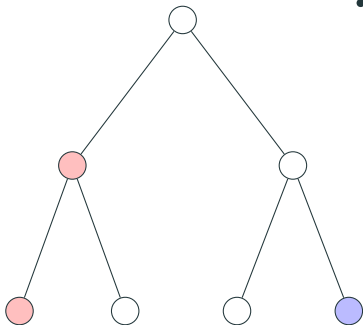
- Bottom-up deterministic tree automaton



Bottom-up tree automata

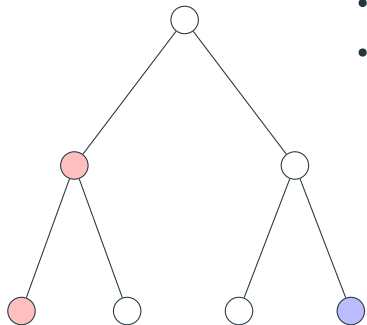
Alphabet Σ : ○ ● ●

- Bottom-up deterministic tree automaton
- States Q : $\{\perp, B, P, T\}$



Bottom-up tree automata

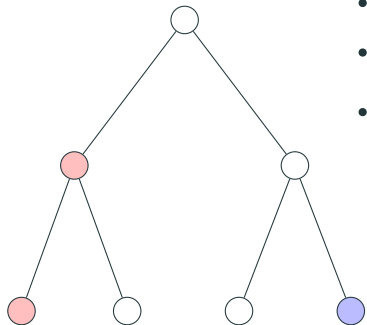
Alphabet Σ : ○ ● ●



- Bottom-up deterministic tree automaton
- States Q : $\{\perp, B, P, T\}$
- Final states $F \subseteq Q$: $\{T\}$

Bottom-up tree automata

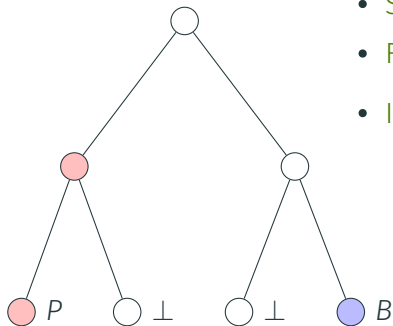
Alphabet Σ : ○ ● ●



- Bottom-up deterministic tree automaton
- States Q : $\{\perp, B, P, T\}$
- Final states $F \subseteq Q$: $\{T\}$
- Initial function ι : ○ \perp ● P ● B

Bottom-up tree automata

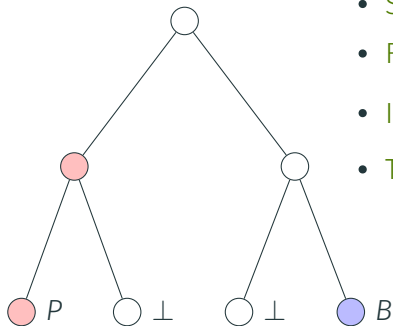
Alphabet Σ : ○ ● ●



- Bottom-up deterministic tree automaton
- States Q : $\{\perp, B, P, T\}$
- Final states $F \subseteq Q$: $\{T\}$
- Initial function ι : ○ \perp ● P ● B

Bottom-up tree automata

Alphabet Σ : ○ ● ●

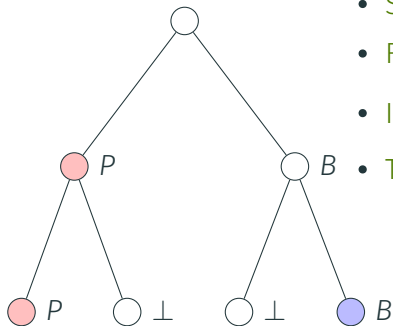


- Bottom-up deterministic tree automaton
- States Q : $\{\perp, B, P, T\}$
- Final states $F \subseteq Q$: $\{T\}$
- Initial function ι : ○ \perp ● P ● B
- Transitions δ (examples):



Bottom-up tree automata

Alphabet Σ : ○ ● ●

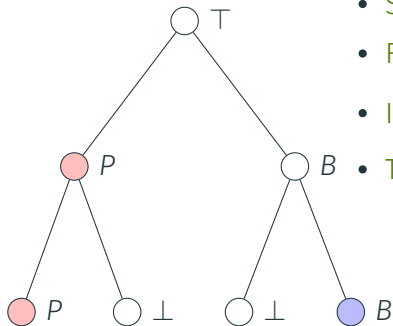


- Bottom-up deterministic tree automaton
- States Q : $\{\perp, B, P, T\}$
- Final states $F \subseteq Q$: $\{T\}$
- Initial function ι : ○ \perp ● P ● B
- Transitions δ (examples):



Bottom-up tree automata

Alphabet Σ : ○ ● ●

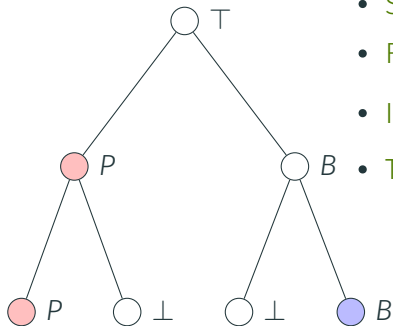


- Bottom-up deterministic tree automaton
- States Q : $\{\perp, B, P, T\}$
- Final states $F \subseteq Q$: $\{T\}$
- Initial function ι : ○ \perp ● P ● B
- Transitions δ (examples):



Bottom-up tree automata

Alphabet Σ : ○ ● ●



- Bottom-up deterministic tree automaton
- States Q : $\{\perp, B, P, T\}$
- Final states $F \subseteq Q$: $\{T\}$
- Initial function ι : ○ \perp ● P ● B
- Transitions δ (examples):



The automaton is...

- deterministic if δ is a function: $\delta : Q^2 \times \Sigma \rightarrow Q$
 - non-deterministic if δ is a relation: $\delta \subseteq Q^2 \times \Sigma \times Q$
- Determinization for bottom-up tree automata (exp blowup).

Example: Boolean (tree) circuit evaluation

- Alphabet: $\Sigma = \{0, 1, \wedge, \vee\}$
- States: $Q = \{q_0, q_1\}$
- Final states: $F = \{q_1\}$
- Initial function:
 - $\iota(0) := q_0$
 - $\iota(1) := q_1$
- Transition function:
 - $\delta(q_i, q_j, \wedge) := q_{i \wedge j}$
 - $\delta(q_i, q_j, \vee) := q_{i \vee j}$

Top-down deterministic tree automata

Like **bottom-up deterministic automata** but differences:

- We add two **dummy leaves** with some special label \perp to all leaves
- The **transition function** is redefined:
 - Now $\delta : Q \times \Sigma \rightarrow Q \times Q$
 - **Semantics**: when we are at state q on a node with label a the state of the automaton on the left and right children is $\delta(q, a)$
- There is now an **initial state** q_0 for the root
- The semantics for **final states** is that **all leaves** should be in a final state

Top-down deterministic tree automata are weaker

- Consider the **finite tree language** containing the two trees:



- It can be accepted by a **bottom-up tree automaton**
- However, with a deterministic top-down tree automaton...
 - Let $(q, q') := \delta(q_0, \bigcirc)$
 - Let $(q_1, q_2) := \delta(q, \text{blue circle})$ and $(q'_1, q'_2) := \delta(q', \text{blue circle})$
 - The first tree is accepted so q_1, q_2 are final
 - The second tree is accepted so q'_1, q'_2 are final
 - Hence the automaton accepts:



Top-down nondeterministic tree automata

- We still add **dummy leaves** with label \perp
- **Alphabet:** Σ
- **States:** Q
- Multiple **initial states** $I \subseteq Q$
- **Final states:** $F \subseteq Q$
- Transition **relation** $\delta \subseteq Q \times \Sigma \times Q^2$

Can be **rewritten** to a bottom-up automaton A' :

- I are the final states of A'
- F defines the initial states of A'
- Reversing δ gives the transition relation of A'

Connections and properties

- **Regular tree language**: accepted by a bottom-up tree automaton
- Equivalently: deterministic bottom-up tree automaton
- Regular tree languages are closed under union, intersection, complement
- Tree automata are equivalent to **monadic second-order** (MSO) logic:
 - **First-order logic**: variables, Boolean connectives \wedge, \vee, \neg , quantifiers \forall, \exists
 - **Quantification over sets (unary predicates)**: $\forall S, \exists S$
 - **Example**: every a -node has a b -descendant:

$$\forall X \left(a(x) \rightarrow \forall X \left((X(x) \wedge \beta_E) \rightarrow (\exists y (X(y) \wedge b(y))) \right) \right)$$

where we have the following (E denotes the edge relation):

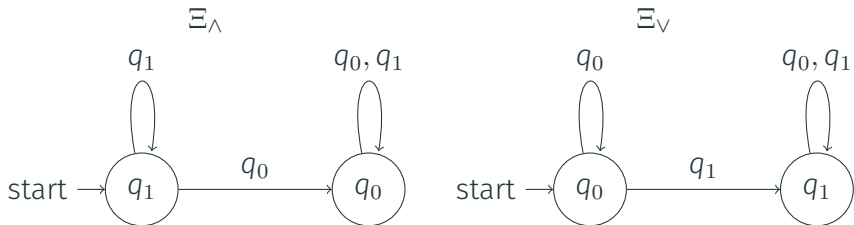
$$\beta_E := \forall yz (E(y, z) \wedge X(y) \rightarrow X(z))$$

Unranked tree automata

- XML documents are **unranked**: each node can have multiple children
- We need specific **automata formalisms** to work on unranked trees
- How can we specify **infinitely many** transition rules?
 - $\delta(q_0, \wedge) = q_0$
 - $\delta(q_0, q_0, \wedge) = q_0$
 - $\delta(q_0, q_0, q_0, \wedge) = q_0$
 - $\delta(q_0, q_0, q_0, q_0, \wedge) = q_0$
 - \vdots
- **Intuition**: define transitions like $\delta((q_0 + q_1)^* q_1 (q_0 + q_1)^*, \vee) = q_1$

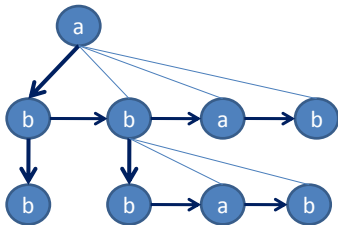
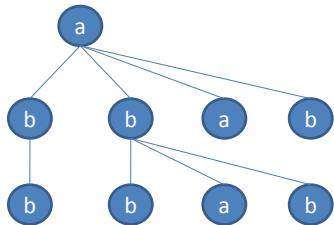
Defining transitions for unranked tree automata

- **Idea:** use an **automaton on words** to define the new state
- We still have the **alphabet** Σ , the **states** Q , the **final states** $F \subseteq Q$, the **initial function** $\iota : \Sigma \rightarrow Q$
- For each $a \in \Sigma$ we have a **word automaton** Ξ_a on the alphabet Q , whose states are also identified with Q



- **Deterministic:** the word automata are deterministic

Building on ranked trees



Ranked tree: FirstChild-NextSibling

F: encoding into a ranked tree

- F is a bijection

F^{-1} : decoding

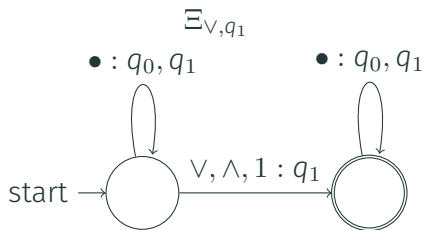
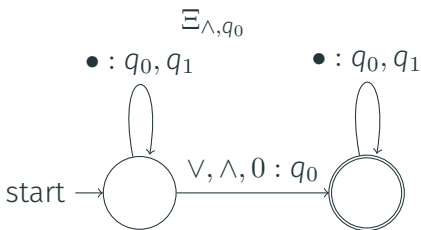
Building on bottom-up ranked trees (2)

- For each Unranked TA A , there is a Ranked TA accepting $F(L(A))$
- For each Ranked TA A , there is an unranked TA accepting $F^{-1}(L(A))$
- Both are easy to construct

Consequence: Unranked TA are closed under union, intersection, complement

Top-down unranked tree automata

- We **adapt** the definition of top-down unranked tree automata
- We have one **initial state** at the root
- For each $a \in \Sigma$ and each $q \in Q$ we have a **word transducer** $\Xi_{a,q}$
- **Semantics**: knowing the label and state of the **parent**, we read the labels of the **children** and we give them a **state**
- To **accept**, the transducers runs must be **accepting**, and all leaves must be labeled with a **final state**



DTD validation

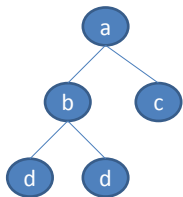
DTDs can be checked with a **deterministic pushdown automaton**:

- The **stack** is used to remember the **nesting** of elements
- While **inside** an element we can check that the word of its children satisfies the **deterministic regular expression** used to define it
- Can be implemented with a **SAX parser**

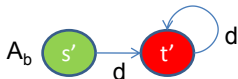
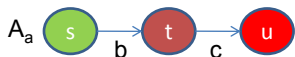
Very efficient validation (2)

<!ELEMENT a (b c) >

<!ELEMENT b (d+) >



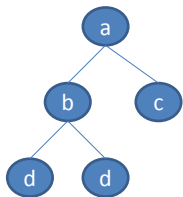
<a><d/><d/><c/>



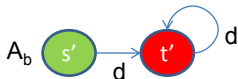
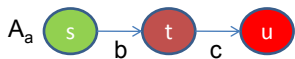
Very efficient validation (2)

`<!ELEMENT a (b c) >`

`<!ELEMENT b (d+) >`



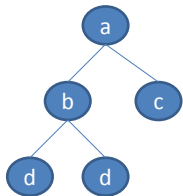
`<a><d/><d/><c/>`



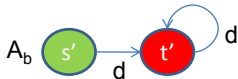
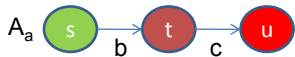
Very efficient validation (2)

<!ELEMENT a (b c) >

<!ELEMENT b (d+) >



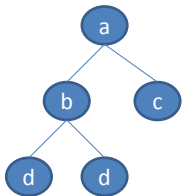
<a><d/><d/><c/>



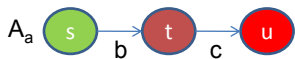
Very efficient validation (2)

<!ELEMENT a (b c) >

<!ELEMENT b (d+) >



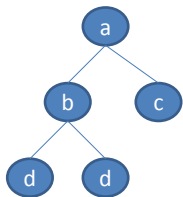
<a><d/><d/><c/>



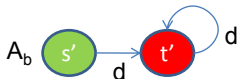
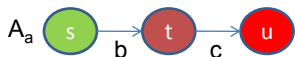
Very efficient validation (2)

<!ELEMENT a (b c) >

<!ELEMENT b (d+) >



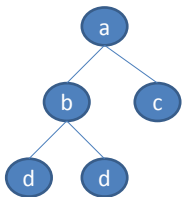
<a><d/><d/><c/>



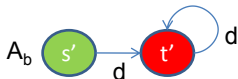
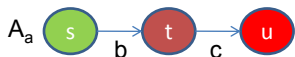
Very efficient validation (2)

<!ELEMENT a (b c) >

<!ELEMENT b (d+) >



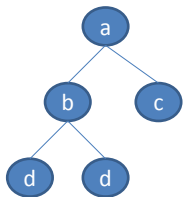
<a><d/><d/><c/>



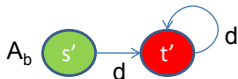
Very efficient validation (2)

<!ELEMENT a (b c) >

<!ELEMENT b (d+) >



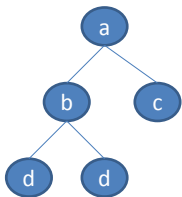
<a><d/><d/><c/>



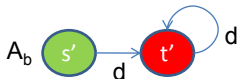
Very efficient validation (2)

<!ELEMENT a (b c) >

<!ELEMENT b (d+) >



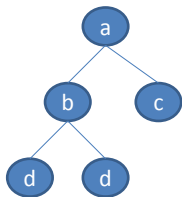
<a><d/><d/><c/>



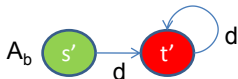
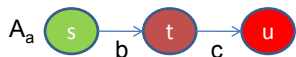
Very efficient validation (2)

<!ELEMENT a (b c) >

<!ELEMENT b (d+) >



<a><d/><d/><c/>



Accept

Warning

- The previous example can be checked with a simple automata on words
- But not the following one

<!ELEMENT part (part*) >

- The stack is needed for accepting

<a>...<a>...


n <a> n

Some bad news for DTD

- Not closed under union

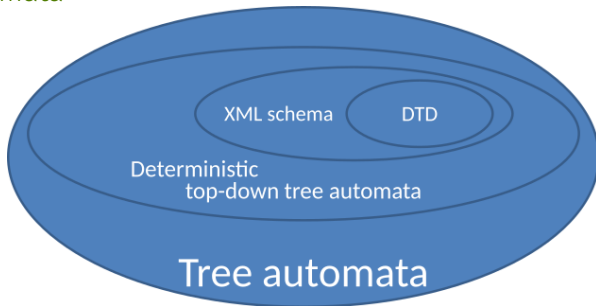
```
DTD1      ...
  <!ELEMENT used( ad* ) >
  <!ELEMENT ad ( year, brand )>

DTD2      ...
  <!ELEMENT new( ad* ) >
  <!ELEMENT ad ( brand )>
```

- $L(\text{DTD1}) \cup L(\text{DTD2})$ cannot be described by a DTD but can be described easily by a tree automata
 - Problem with the type of **ad** that depends of its parent
- Also not closed under complement
- Limited expressive power

XML Schema validation for version 1.0

- The strange requirements of **XSD version 1.0** make it a subset of the languages that can be checked with **deterministic top-down tree automata**



- With version 1.1, the use of **assertions** means that the tree language is no longer **regular**:

$\text{count}(\//a) = \text{count}(\//b)$ and $\text{count}(\//b) = \text{count}(\//c)$

Sources

- Slide 11–14: <https://www.irif.fr/~carton/Enseignement/XML/Cours/DTD/>
- Slide 17: <http://xmlfr.org/documentations/tutoriels/001218-0001>
- Slide 24: https://en.wikipedia.org/wiki/RELAX_NG
- Slide 25: <https://www.xml.com/pub/a/2003/11/12/schematron.html>