A Mini Course on
Trees, Automata and
XML

Paris
June 2004

Thomas
Schwentick
Why XML?
Three Questions

Question 1

Why XML?

Answer

Have a look into the ≥ 20 XML papers at SIGMOD/PODS
Why Trees?
Why Trees?

Look at this:

```xml
<Composer>
  <Name> Claude Debussy </Name>
  <Vita>
    <Born> <When> August 22, 1862 </When> <Where> Paris </Where> </Born>
    <Married> <When> October 1899 </When> <Whom> Rosalie </Whom> </Married>
    <Married> <When> January 1908 </When> <Whom> Emma </Whom> </Married>
    <Died> <When> March 25, 1918 </When> <Where> Paris </Where> </Died>
  </Vita>
  <Piece>
    <PTitle> La Mer </PTitle>
    <PYear> 1905 </PYear>
    <Instruments> Large orchestra </Instruments>
    <Movements> 3 </Movements>
    ...
  </Piece>
  ...
</Composer>
```

PODS 2004 Thomas Schwentick Trees, Automata & XML
Why Trees?

Look at this:

Composer

Name

Claude Debussy

Born

When

1862

Where

Paris

Married

When

1899

Whom

Rosalie

Married

When

1908

Whom

Emma

Died

When

1918

Where

Paris

Piece

PTitle

La Mer

PYear

1905

Instruments

Large orchestra

Movements

3
Why Automata?
Why Automata?

Answer

That’s our topic for the remaining 88 minutes
Question: Why is XML appealing for Theory people?
Question: Why is XML appealing for Theory people?

Years ago...

- Theoretical Computer Science for Database Theorists: Logics, Complexity, Algorithms, ...
- Database Theory for Theoretical Computer Scientists:
Question: Why is XML appealing for Theory people?

Years ago...

- Theoretical Computer Science for Database Theorists: Logics, Complexity, Algorithms, ...
- Database Theory for Theoretical Computer Scientists: terra incognita
Question: Why is XML appealing for Theory people?

Years ago...
- Theoretical Computer Science for Database Theorists: Logics, Complexity, Algorithms, ...
- Database Theory for Theoretical Computer Scientists: terra incognita

After the advent of XML
Many connections between Formal Languages & Automata Theory and XML & Database Theory
Question: Why trees?
Question: Why trees?

A Natural Answer

- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based
Question: Why trees?

A Natural Answer
- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

Limitations
- But trees cannot model all aspects of XML (e.g., IDREFs, data values)
  ⇒ Sometimes extensions are needed
- E.g., directed graphs instead of trees
**Question:** Why trees?

**A Natural Answer**
- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

**Limitations**
- But trees cannot model all aspects of XML (e.g., IDREFs, data values)
  - Sometimes extensions are needed
- E.g., directed graphs instead of trees

---

**Example**

![Diagram of an XML tree structure](image-url)
Question: Why trees?

A Natural Answer
- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

Limitations
- But trees cannot model all aspects of XML (e.g., IDREFs, data values)
  ⇒ Sometimes extensions are needed
- E.g., directed graphs instead of trees

Example
Question: Why trees?

A Natural Answer
- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree-based

Limitations
- But trees cannot model all aspects of XML (e.g., IDREFs, data values)
  - Sometimes extensions are needed
- E.g., directed graphs instead of trees

Nevertheless
In this tutorial we will concentrate on the tree view at XML
Question: Why automata?

Ingredients of XML
Concepts from formal languages are obviously present around XML:

- Labelled trees
- DTD: context-free grammars
- DTD: regular expressions
- XPath: regular path expressions

We will see
Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation
Question: Why automata?

Ingredients of XML

Concepts from formal languages are obviously present around XML:

- Labelled trees
- DTD: context-free grammars
- DTD: regular expressions
- XPath: regular path expressions

We will see

Automata turn out to be useful as:

→ a means to define robust classes with clear semantics
- a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation
We will see
Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation

Ingredients of XML
Concepts from formal languages are obviously present around XML:

- Labelled trees
- DTD: context-free grammars
- DTD: regular expressions
- XPath: regular path expressions

Question: Why automata?
Question: Why automata?

Ingredients of XML
Concepts from formal languages are obviously present around XML:

- Labelled trees
- DTD: context-free grammars
- DTD: regular expressions
- XPath: regular path expressions

We will see
Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation
Question: Why automata?

**Ingredients of XML**

Concepts from formal languages are obviously present around XML:

- Labelled trees
- DTD: context-free grammars
- DTD: regular expressions
- **XPath**: regular path expressions

**We will see**

Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation
Four important kinds of XML processing

Validation
Check whether an XML document is of a given type

Navigation
Select a set of positions in an XML document

Querying
Extract information from an XML document

Transformation
Construct a new XML document from a given one
Four important kinds of XML processing .......... and their languages

Validation

Check whether an XML document is of a given type

DTD, XML Schema

Navigation

Select a set of positions in an XML document

XPath

Querying

Extract information from an XML document

XQuery

Transformation

Construct a new XML document from a given one

XSLT
Example document

```
<Composer>
  <Name> Claude Debussy </Name>
  <Vita>
    <Born> <When> August 22, 1862 </When> <Where> Paris </Where> </Born>
    <Married> <When> October 1899 </When> <Whom> Rosalie </Whom> </Married>
    <Married> <When> January 1908 </When> <Whom> Emma </Whom> </Married>
    <Died> <When> March 25, 1918 </When> <Where> Paris </Where> </Died>
  </Vita>
  <Piece>
    <PTitle> La Mer </PTitle>
    <PYear> 1905 </PYear>
    <Instruments> Large orchestra </Instruments>
    <Movements> 3 </Movements>
    ...
  </Piece>
  ...
</Composer>
```
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born><When>August 22, 1862</When><Where>Paris</Where></Born>
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>
    < Died><When>March 25, 1918</When><Where>Paris</Where></Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
  ...
  </Piece>
  ...
</Composer>
```
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born>August 22, 1862</Born>
    <Where>Paris</Where>
    <When>October 1899</When>
    <Whom>Rosalie</Whom>
    <Married>January 1908</Married>
    <Whom>Emma</Whom>
    <Died>March 25, 1918</Died>
    <Where>Paris</Where>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```

Example

```xml
<!DOCTYPE Composers [ 
  <!ELEMENT Composers (Composer*)> 
  <!ELEMENT Composer (Name, Vita, Piece*)> 
  <!ELEMENT Vita (Born, Married*, Died?)> 
  <!ELEMENT Born (When, Where)> 
  <!ELEMENT Married (When, Whom)> 
  <!ELEMENT Died (When, Where)> 
  <!ELEMENT Piece (PTitle, PYear, 
       Instruments, Movements)> 
]>
```
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born>August 22, 1862</Born>
    <Where>Paris</Where>
    <When>August 22, 1862</When>
    <When>Paris</Where>
    <Born>August 22, 1862</Born>
    <When>April 22, 1862</When>
    <Where>Paris</Where>
    < Married>October 1899</Married>
    <When>October 1899</When>
    <Whom>Rosalie</Whom>
    <Married>October 1899</Married>
    <When>October 1899</When>
    <Whom>Rosalie</Whom>
    < Married>January 1908</Married>
    <When>January 1908</When>
    <Whom>Emma</Whom>
    <Married>January 1908</Married>
    <When>January 1908</When>
    <Whom>Emma</Whom>
    < Died>March 25, 1918</Died>
    <When>March 25, 1918</When>
    <Where>Paris</Where>
    <Died>March 25, 1918</Died>
    <Where>Paris</Where>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```
XPath

XPath expressions select sets of nodes of XML documents by specifying navigational patterns.
Intro  XML Processing  Navigation: XPath

XPath

XPath expressions select sets of nodes of XML documents by specifying navigational patterns

Example document

```xml
<Composer>
   <Name>Claude Debussy</Name>
   <Vita>
      <Born>August 22, 1862</Born>
      <Where>Paris</Where>
      <When>October 1899</When>
      <Whom>Rosalie</Whom>
      <Married>January 1908</Married>
      <Whom>Emma</Whom>
      <Died>March 25, 1918</Died>
      <Where>Paris</Where>
   </Vita>
   <Piece>
      <PTitle>La Mer</PTitle>
      <PYear>1905</PYear>
      <Instruments>Large orchestra</Instruments>
      <Movements>3</Movements>
      ...
   </Piece>
   ...
</Composer>
```

Example query

```
//Vita/Died/*
```
XPath

XPath expressions select sets of nodes of XML documents by specifying navigational patterns.

Example query

//Vita/Died/*
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born>August 22, 1862</Born>
    <Where>Paris</Where>
    <When>August 22, 1862</When>
    <When>Paris</When>
    <Born>August 22, 1862</Born>
    <Married>October 1899</Married>
    <Where>Rosalie</Where>
    <When>October 1899</When>
    <When>Rosalie</When>
    <Married>October 1899</Married>
    <Married>January 1908</Married>
    <Where>Emma</Where>
    <When>January 1908</When>
    <When>Emma</When>
    <Married>January 1908</Married>
    <Died>March 25, 1918</Died>
    <Where>Paris</Where>
    <When>March 25, 1918</When>
    <When>Paris</When>
    <Died>March 25, 1918</Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```
**Example document**

```
<Composer>
  <Name> Claude Debussy </Name>
  <Vita>
    <Born> <When> August 22, 1862 </When> <Where> Paris </Where> </Born>
    <Married> <When> October 1899 </When> <Whom> Rosalie </Whom> </Married>
    <Married> <When> January 1908 </When> <Whom> Emma </Whom> </Married>
    <Died> <When> March 25, 1918 </When> <Where> Paris </Where> </Died>
  </Vita>
  <Piece>
    <PTitle> La Mer </PTitle>
    <PYear> 1905 </PYear>
    <Instruments> Large orchestra </Instruments>
    <Movements> 3 </Movements>
    ...
  </Piece>
  ...
</Composer>
```

---

**XQuery**

XQuery is a full-fledged XML query language.
XQuery is a full-fledged XML query language

Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born><When>August 22, 1862</When><Where>Paris</Where></Born>
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>
    <Died><When>March 25, 1918</When><Where>Paris</Where></Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```

Example query

```xml
for $x in doc('composers.xml')/Composer
  where $x/Vita/Died/Where = 'Paris'
  return $x/Name
```
XQuery is a full-fledged XML query language.

Example query:

```xquery
for $x in doc('composers.xml')/Composer
where $x/Vita/Died/Where = 'Paris'
return $x/Name
```

---

Result:

- Claude Debussy
- Eric Satie
- Hector Berlioz
- Camille Saint-Saëns
- Frédéric Chopin
- Maurice Ravel
- Jim Morrison
- César Franck
- Gabriel Fauré
- George Bizet
- ...
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born>August 22, 1862</Born>
    <Where>Paris</Where>
    <Married>October 1899</Married>
    <Whom>Rosalie</Whom>
    <Married>January 1908</Married>
    <Whom>Emma</Whom>
    <Died>March 25, 1918</Died>
    <Where>Paris</Where>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTtitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```

PODS 2004 Thomas Schwentick Trees, Automata & XML
XSLT transforms documents by means of templates

Example document

```
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born><When>August 22, 1862</When><Where>Paris</Where></Born>
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>
    <Died><When>March 25, 1918</When><Where>Paris</Where></Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```
XSLT transforms documents by means of templates

Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born>August 22, 1862</Born>
    <Where>Paris</Where>
    <When>When</When>
    <Where>Where</Where>
    <When>Born</When>
    <Married>October 1899</Married>
    <When>When</When>
    <Whom>Rosalie</Whom>
    <When>When</When>
    <Whom>Married</Whom>
    <Married>January 1908</Married>
    <When>When</When>
    <Whom>Emma</Whom>
    <When>When</When>
    <Whom>Married</Whom>
    <Died>March 25, 1918</Died>
    <When>When</When>
    <Where>Paris</Where>
    <When>When</When>
    <Where>Died</Where>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTtitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```

Example

```xml
<xsl:template match="Composer[Vita/Where='Paris']">
  <ParisComposer>
    <xsl:copy-of select="Name"/>
    <xsl:copy-of select="Vita/Born"/>
  </ParisComposer>
</xsl:template>
```
**XSLT**

XSLT transforms documents by means of templates.

**Example**

```
<xsl:template match="Composer[Vita//Where='Paris']">
  <ParisComposer>
    <Name>Claude Debussy</Name>
    <Born>
      <When>August 22, 1862</When>
      <Where>Paris</Where>
    </Born>
  </ParisComposer>
</xsl:template>
```

```
<xsl:template match="Composer[Marrried]">
  <ParisComposer>
    <Name>Frédéric Chopin</Name>
    <Born>
      <When>March 1, 1810</When>
      <Where>Żelazowa</Where>
    </Born>
  </ParisComposer>
</xsl:template>
```

```
<xsl:template match="Composer[Married]">
  <ParisComposer>
    <Name>Camille Saint-Saëns</Name>
    <Born>
      <When>October 9, 1835</When>
      <Where>Paris</Where>
    </Born>
  </ParisComposer>
</xsl:template>
```
Aim

- Introduction
- Basic techniques and models
- Not a survey
- In particular: many important papers are not mentioned

Overall structure

Part 1: Background on tree automata and how they can be adapted for XML purposes

Part 2: Examples for the use of automata for XML
- Two robust classes of schema languages
- A robust class of node-selecting queries
- Automata as an algorithmic tool for checking XPath query containment

Part 3: Some words about related results and about extensions and limitations
A String

abcab
A String
abcab

String as Tree

a
  b
  c
  a
  b
A String

abcab

String as Tree

A Ranked Tree

 PODS 2004 Thomas Schwentick  Trees, Automata & XML
Pre-XML From Strings to Trees

A String
abcab

String as Tree
a
 b
 c
 a
 b

A Ranked Tree
a
 b
 c

An Unranked Tree
a
 . . .
 e c e a
 b
 c
e c c c
ea
 a
e c e e
e c e
a

PODS 2004 Thomas Schwentick Trees, Automata & XML 16
XML and Trees

- XML trees are **unranked**: the number of children of a node is not restricted
- Automata have first been considered on **ranked** trees, where each symbol has a fixed number of children (rank)
- Most important ideas were already developed for ranked trees

→ Let us take a look at this first
Question

How do automata generalize to trees?

Sequential

Parallel
From String Automata to Tree Automata

Question
How do automata generalize to trees?

Sequential

```
      a
     /|
    e c
   /|
  a e
 /|
e a
```

Parallel

```
      a
     /|
    e c
   /|
  a e
 /|
e c
```

PODS 2004
Thomas Schwentick
Trees, Automata & XML
Question
How do automata generalize to trees?

Sequential

Parallel
From String Automata to Tree Automata

Question
How do automata generalize to trees?

Sequential

Parallel
Question
How do automata generalize to trees?

Sequential

Parallel
Question

How do automata generalize to trees?

Sequential

Parallel
Question

How do automata generalize to trees?

Sequential

Parallel
From String Automata to Tree Automata

Question
How do automata generalize to trees?

Sequential

```
    a
   / 
  e   e
 /    |
c  e   a
 / 
 e  a
```

Parallel

```
    a
   / 
  e   e
 /    |
c  e   a
 / 
 e  a
```

PODS 2004 Thomas Schwentick Trees, Automata & XML
Question

How do automata generalize to trees?

Sequential

Parallel
Question

How do automata generalize to trees?

Sequential

Parallel
Question

How do automata generalize to trees?

Sequential

Parallel
Question
How do automata generalize to trees?

Sequential

Parallel
Question
How do automata generalize to trees?

Sequential

Parallel
Question

How do automata generalize to trees?

Sequential

Parallel
Question
How do automata generalize to trees?

Sequential

Parallel
How do automata generalize to trees?

Sequential

Parallel
How do automata generalize to trees?

**Sequential**

```
  a
 /   \
 e    e
 /     /
 c     c
 /     /  \
 a     a   e
 /     \
 e     a
```

**Parallel**

```
  a
 /   \
 e    e
 /     /
 c     c
 /     /  \
 a     a   e
 /     \
 e     a
```

PODS 2004  Thomas Schwentick  Trees, Automata & XML
Example: Tree-structured Boolean Circuits

Idea
- Tree-structured Boolean circuits
- Two states: \( q_0, q_1 \)
- Accepting at the root: \( q_1 \)

Transitions
- \( \delta(\land, q_1) = \{(q_1, q_1)\} \)
- \( \delta(\land, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \)
- \( \delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \)
- \( \delta(\lor, q_0) = \{(q_0, q_0)\} \)
- \( \delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset \)
- \( \delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset \)
Example: Tree-structured Boolean Circuits

Idea
Tree-structured Boolean circuits
Two states: \( q_0, q_1 \)
Accepting at the root: \( q_1 \)

Transitions
\[
\begin{align*}
\delta(\wedge, q_1) &= \{(q_1, q_1)\} \\
\delta(\wedge, q_0) &= \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \\
\delta(\vee, q_1) &= \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \\
\delta(\vee, q_0) &= \{(q_0, q_0)\} \\
\delta(0, q_0) &= \{\varepsilon\}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{\varepsilon\}; \delta(1, q_0) = \emptyset
\end{align*}
\]
**Example: Tree-structured Boolean Circuits**

Idea

Tree-structured Boolean circuits

Two states: $q_0, q_1$

Accepting at the root: $q_1$

**Transitions**

- $\delta(\land, q_1) = \{(q_1, q_1)\}$
- $\delta(\land, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$
- $\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$
- $\delta(\lor, q_0) = \{(q_0, q_0)\}$
- $\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$
- $\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$
Example: Tree-structured Boolean Circuits

Idea
Tree-structured Boolean circuits
Two states: \( q_0, q_1 \)
Accepting at the root: \( q_1 \)

Transitions
\[
\begin{align*}
\delta(\wedge, q_1) &= \{(q_1, q_1)\} \\
\delta(\wedge, q_0) &= \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \\
\delta(\vee, q_1) &= \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \\
\delta(\vee, q_0) &= \{(q_0, q_0)\} \\
\delta(0, q_0) &= \{\epsilon\}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{\epsilon\}; \delta(1, q_0) = \emptyset
\end{align*}
\]
Idea
Tree-structured Boolean circuits
Two states: $q_0, q_1$
Accepting at the root: $q_1$

Transitions
\[
\begin{align*}
\delta(\land, q_1) &= \{ (q_1, q_1) \} \\
\delta(\land, q_0) &= \{ (q_0, q_1), (q_1, q_0), (q_0, q_0) \} \\
\delta(\lor, q_1) &= \{ (q_0, q_1), (q_1, q_0), (q_1, q_1) \} \\
\delta(\lor, q_0) &= \{ (q_0, q_0) \} \\
\delta(0, q_0) &= \{ \epsilon \}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{ \epsilon \}; \delta(1, q_0) = \emptyset
\end{align*}
\]
Pre-XML
Parallel Tree Automata
Non-det. Top-Down Automata

Example

Idea
Guess the correct values starting from the root
Check at the leaves
Three states: $q_0, q_1, \text{acc}$
Initial state $q_1$ at the root
Accepting if all leaves end in $\text{acc}$

Transitions
\[
\delta(\wedge, q_1) = \{(q_1, q_1)\}
\]
\[
\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}
\]
\[
\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}
\]
\[
\delta(\vee, q_0) = \{(q_0, q_0)\}
\]
\[
\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset
\]
\[
\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset
\]
Idea
Guess the correct values starting from the root
Check at the leaves
Three states: $q_0, q_1, \text{acc}$
Initial state $q_1$ at the root
Accepting if all leaves end in $\text{acc}$

Transitions
\[
\begin{align*}
\delta(\wedge, q_1) &= \{(q_1, q_1)\} \\
\delta(\wedge, q_0) &= \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \\
\delta(\vee, q_1) &= \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \\
\delta(\vee, q_0) &= \{(q_0, q_0)\} \\
\delta(0, q_0) &= \{\text{acc}\}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{\text{acc}\}; \delta(1, q_0) = \emptyset
\end{align*}
\]
Idea
Guess the correct values starting from the root
Check at the leaves
Three states: $q_0$, $q_1$, acc
Initial state $q_1$ at the root
Accepting if all leaves end in acc

Transitions
\begin{align*}
\delta(\land, q_1) &= \{(q_1, q_1)\} \\
\delta(\land, q_0) &= \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \\
\delta(\lor, q_1) &= \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \\
\delta(\lor, q_0) &= \{(q_0, q_0)\} \\
\delta(0, q_0) &= \{\text{acc}\}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{\text{acc}\}; \delta(1, q_0) = \emptyset
\end{align*}
Example

\[
\begin{align*}
\land q_1 & \\
\lor q_1 & \\
\land q_0 & \\
0 & \\
1 & \\
\lor q_1 & \\
1 & \\
0 & \\
\land q_1 & \\
\lor q_1 & \\
\land q_1 & \\
0 & \\
1 & \\
1 & \\
1 & \\
\end{align*}
\]

Idea

Guess the correct values starting from the root
Check at the leaves
Three states: \( q_0, q_1, \text{acc} \)
Initial state \( q_1 \) at the root
Accepting if all leaves end in \( \text{acc} \)

Transitions

\[
\begin{align*}
\delta(\land, q_1) &= \{(q_1, q_1)\} \\
\delta(\land, q_0) &= \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \\
\delta(\lor, q_1) &= \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \\
\delta(\lor, q_0) &= \{(q_0, q_0)\} \\
\delta(0, q_0) &= \{\text{acc}\}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{\text{acc}\}; \delta(1, q_0) = \emptyset
\end{align*}
\]
Example

\[ \begin{array}{c}
q_0 \\
\wedge \end{array} \quad \begin{cases}
q_1 \\
\vee \end{cases} \quad \begin{array}{c}
q_0 \\
\wedge \end{array} \quad \begin{cases}
q_1 \\
\vee \end{cases} \quad \begin{array}{c}
q_0 \\
\wedge \end{array} \quad \begin{cases}
q_1 \\
\vee \end{cases} \quad \begin{array}{c}
q_0 \\
\wedge \end{array} \quad q_1
\end{array} \]

Idea

Guess the correct values starting from the root
Check at the leaves
Three states: \( q_0, q_1, \text{acc} \)
Initial state \( q_1 \) at the root
Accepting if all leaves end in \( \text{acc} \)

Transitions

\[ \begin{align*}
\delta(\wedge, q_1) &= \{ (q_1, q_1) \} \\
\delta(\wedge, q_0) &= \{ (q_0, q_1), (q_1, q_0), (q_0, q_0) \} \\
\delta(\vee, q_1) &= \{ (q_0, q_1), (q_1, q_0), (q_1, q_1) \} \\
\delta(\vee, q_0) &= \{ (q_0, q_0) \} \\
\delta(0, q_0) &= \{ \text{acc} \}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{ \text{acc} \}; \delta(1, q_0) = \emptyset
\end{align*} \]
Idea

Guess the correct values starting from the root
Check at the leaves
Three states: $q_0, q_1, \text{acc}$
Initial state $q_1$ at the root
Accepting if all leaves end in acc

Transitions

$\delta(\land, q_1) = \{(q_1, q_1)\}$
$\delta(\land, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$
$\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$
$\delta(\lor, q_0) = \{(q_0, q_0)\}$
$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$
$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$
Definition

A bottom-up automaton is deterministic if for each \( a \) and \( p \neq q \):
\[
\delta(a, p) \cap \delta(a, q) = \emptyset
\]

Theorem

The following are equivalent for a tree language \( L \):

(a) \( L \) is accepted by a nondeterministic bottom-up automaton

(b) \( L \) is accepted by a deterministic bottom-up automaton

(c) \( L \) is accepted by a nondeterministic top-down automaton

Proof Idea

(a) \( \implies \) (b): Powerset construction

(a) \( \iff \) (c): Just the same thing, viewed in two different ways
Observation

- \((q_0, q_1) \in \delta(\lor, q_1)\) can be interpreted as an allowed pattern:

- A tree is accepted, iff there is a labelling with states such that
  - all local patterns are allowed
  - the root is labelled with \(q_1\)
Observation

- \((q_0, q_1) \in \delta(\lor, q_1)\) can be interpreted as an allowed pattern:

- A tree is accepted, iff there is a labelling with states such that
  - all local patterns are allowed
  - the root is labelled with \(q_1\)

Example
Observation

- \((q_0, q_1) \in \delta(\vee, q_1)\) can be interpreted as an allowed pattern:

- A tree is accepted, iff there is a labelling with states such that
  - all local patterns are allowed
  - the root is labelled with \(q_1\)

Example
**Definition (MSO logic)**

- **Formulas** talk about
  - edges of the tree ($E$)
  - node labels ($Q_0, Q_1, Q^\land, Q^\lor$)
  - the root of the tree (root)

- **First-order-variables** represent nodes

- **Monadic second-order** (MSO) variables represent sets of nodes

**Example: Boolean Circuits**

- **Boolean circuit true**

  $\equiv \exists X \ X(\text{root}) \land \forall x$

  $\quad (Q_0(x) \rightarrow \neg X(x)) \land$

  $\quad ((Q^\land(x) \land X(x)) \rightarrow (\forall y\ [E(x, y) \rightarrow X(y)])) \land$

  $\quad ((Q^\lor(x) \land X(x)) \rightarrow (\exists y\ [E(x, y) \land X(y)]))$

**Theorem (Doner 1970; Thatcher, Wright 1968)**

MSO $\equiv$ Regular Tree Languages
Theorem

MSO \equiv \text{Regular Tree Languages}

Proof Idea

**Automata } MSO:**
Formula expresses that there exists a correct tiling

**MSO } Automata:** more involved

Basic idea:
Automaton computes for each node \( v \) the set of formulas which hold in the subtree rooted at \( v \)
Formula $\Rightarrow$ automaton

- Let $\varphi$ be an MSO-formula, $k := \text{quantifier-depth of } \varphi$

- $k$-type of a tree $t :=$ (essentially)
  set of MSO-formulas $\psi$ of quantifier-depth $\leq k$ which hold in $t$

- $t_1 \equiv_k t_2 : k$-type$(t_1) = k$-type$(t_2)$

- Automaton computes $k$-type of tree and concludes whether $\varphi$ holds

Crucial fact:

$\triangleleft\quad t_1 \equiv_k t_1' \quad \triangleleft$

$\triangleleft\quad t_2 \equiv_k t_2' \quad \triangleleft$

$\quad \quad \quad \quad \Rightarrow$

$\quad \triangleleft\quad t_1 \quad t_2 \quad \equiv_k t_1' \quad t_2'$

$\quad \quad \quad \quad \Rightarrow$

$\quad \triangleleft\quad l \quad \downarrow$

$\quad \triangleleft\quad t_1 \quad l \quad t_2 \quad \equiv_k t_1' \quad t_2'$
Question
What is the right notion for deterministic top-down automata?

3 Possibilities
State at a node $v$ might depend on

- state and symbol of parent
- state and symbol of parent and symbol of $v$
- state and symbol of parent and symbols at $v$ and its sibling
### Question

What is a good acceptance mechanism for deterministic top-down automata?

### Several possibilities

1. At all leaves states have to be accepting
2. There is a leave with an accepting state

(2) is problematic for complement and intersection
(1) is problematic for complement and union
**Definition** (Root-to-frontier automata with regular acceptance condition)

- Tree automata $\mathcal{A}$ are equipped with an additional regular string language $L$ over $Q \times \Sigma$
- $\mathcal{A}$ accepts $t$ if the (state,symbol)-string at the leaves (from left to right) is in $L$

**Illustration**

A robust class

- The resulting class is closed under Boolean operations
- Good algorithmic properties
- Does not capture all regular tree languages
Regular tree languages

- Regular tree languages are a robust class
- Characterized by
  - parallel tree automata
  - MSO logic
  - several other models
- They are the natural analog of regular string languages
- Deterministic top-down automata with regular acceptance conditions define a weaker but also robust class
I
- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML
- Parallel Tree Automata
- Sequential Tree Automata
- Decision Problems

II
- Schemas
- Node Selecting Queries
- XPath Query Containment

III
- Extensions
- Conclusion
Definition (Tree-walk automata)

Depending on

- current state
- symbol of current node
- position of current node wrt its siblings

the automaton moves to a neighbor and takes a new state

Question

What is the expressive power of tree-walk automata?
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

$q^0$

$q^1$
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

0
1
1
0
0
1
1
1
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

```
q^0
\wedge
V
\wedge
\wedge
^q_
V
V
\wedge
\wedge
1
1
1
0
0
1
1
1
```

Idea
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

PODS 2004 Thomas Schwentick Trees, Automata & XML
**Fact**
- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

**Idea**

```
0 1 1 0 0 1 1 1
```

```
0
```

```
1
```

```
q^1
```

```
q^0
```

```
^q_
```

```
_0^q
```

```
^1q
```

```
_1q
```

```
0
```

```
1
```

```
V
```

```
^q
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```

```
V
```
**Fact**
- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

**Example**

**Idea**

$q^0$

$q^1$

PODS 2004 Thomas Schwentick Trees, Automata & XML
Fact
- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
$q^0$
$q^1$

PODS 2004
Thomas Schwentick
Trees, Automata & XML
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

PODS 2004  Thomas Schwentick  Trees, Automata & XML
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

PODS 2004  Thomas Schwentick  Trees, Automata & XML
Fact

• Tree-walk automata can evaluate Boolean circuit trees

• 5 states

Example

Idea

$q^0$

$q^1$
Fact
- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

PODS 2004
Thomas Schwentick
Trees, Automata & XML
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
Fact
- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

$\wedge^0$

$\vee^1$

$\wedge^1$

$\vee$

$\wedge$

$\wedge$

$\vee$

$\wedge$

$\wedge$

$\vee$

$\wedge$

$\wedge$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$

$\wedge$

$\vee$
Fact
- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

PODS 2004 Thomas Schwentick Trees, Automata & XML
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

\( q^0 \)

\( q^1 \)
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

PODS 2004 Thomas Schwentick Trees, Automata & XML
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

\[ q^0 \quad q^1 \]

Trees, Automata & XML
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

\[ \begin{array}{c}
q_0 \\
\land \\
\lor \\
\land \\
\lor \\
\land \end{array} \]

\[ \begin{array}{c}
0 \\
1 \\
1 \\
0 \\
0 \\
1 \\
1 \\
1 \end{array} \]

Idea

\[ q^0 \]

\[ q^1 \]
Fact
- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea

$q^0$

$q^1$
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example

Idea
Theorem (Bojanczyk, Colcombet 2004)
Deterministic TWAs are weaker than nondeterministic TWAs

Corollary
Deterministic TWAs do not capture all regular tree languages

Conjecture
Nondeterministic TWAs do not capture all regular tree languages
Sequential Tree Automata

Overview of Models

Non-det. top-down tree automata
Non-det. bottom-up tree automata
Det. bottom-up tree automata

Det. top-down tree automata

Non-det. tree walk automata

Det. tree walk automata
I
- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II
- Schemas
- Node Selecting Queries
- XPath Query Containment

III
- Extensions
- Conclusion

Parallel Tree Automata
Sequential Tree Automata
Decision Problems

PODS 2004
Thomas Schwentick
Trees, Automata & XML
Algorithmic problems
- We consider the following algorithmic problems
- All of them will be useful in the XML context

<table>
<thead>
<tr>
<th>Membership test for $A$</th>
<th>Membership test (combined)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Given:</strong> Tree $t$</td>
<td><strong>Given:</strong> Tree Automaton $A$, tree $t$</td>
</tr>
<tr>
<td><strong>Question:</strong> Is $t \in L(A)$?</td>
<td><strong>Question:</strong> Is $t \in L(A)$?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-emptiness</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Given:</strong> Automaton $A$</td>
</tr>
<tr>
<td><strong>Question:</strong> Is $L(A) \neq \emptyset$?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Containment</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Given:</strong> Automata $A_1, A_2$</td>
<td><strong>Given:</strong> Automata $A_1, A_2$</td>
</tr>
<tr>
<td><strong>Question:</strong> Is $L(A_1) \subseteq L(A_2)$?</td>
<td><strong>Question:</strong> Is $L(A_1) = L(A_2)$?</td>
</tr>
</tbody>
</table>
Time Bounds for the combined complexity of membership test for tree automata:

- Deterministic (parallel) tree automata: \( O(|A||t|) \)
- Nondeterministic (parallel) tree automata: \( O(|A|^2|t|) \)
  (Compute, for each node, the set of reachable states)
- Deterministic TWAs: \( O(|A|^2|t|) \)
  (Compute, for each node \( v \), the aggregated behavior of \( A \) on its subtree: **Behavior function**)
- Nondeterministic TWAs: \( O(|A|^3|t|) \)
  (Compute, for each node \( v \), the aggregated behavior of \( A \) on its subtree: **Behavior relation**)

**Facts**

Pre-XML  Decision Problems  Membership Test
### Facts

Time Bounds for the combined complexity of membership test for tree automata:

- **Deterministic (parallel) tree automata:** $O(|A||t|)$
  - (Compute, for each node, the set of reachable states)

- **Nondeterministic (parallel) tree automata:** $O(|A|^2|t|)$
  - (Compute, for each node, the aggregated behavior of $A$ on its subtree: **Behavior function**)

- **Deterministic TWAs:** $O(|A|^2|t|)$
  - (Compute, for each node $v$, the aggregated behavior of $A$ on its subtree: **Behavior function**)

- **Nondeterministic TWAs:** $O(|A|^3|t|)$
  - (Compute, for each node $v$, the aggregated behavior of $A$ on its subtree: **Behavior relation**)

---

PODS 2004  
Thomas Schwentick  
Trees, Automata & XML
Facts

Time Bounds for the combined complexity of membership test for tree automata:

- Deterministic (parallel) tree automata: $O(|A||t|)$
- Nondeterministic (parallel) tree automata: $O(|A|^2|t|)$
  (Compute, for each node, the set of reachable states)
- Deterministic TWAs: $O(|A|^2|t|)$
  (Compute, for each node $v$, the aggregated behavior of $A$ on its subtree: **Behavior function**)
- Nondeterministic TWAs: $O(|A|^3|t|)$
  (Compute, for each node $v$, the aggregated behavior of $A$ on its subtree: **Behavior relation**)
Facts

Time Bounds for the combined complexity of membership test for tree automata:

- Deterministic (parallel) tree automata:
  \[ O(|A|^2|t|) \]
  (Compute, for each node, the set of reachable states)

- Deterministic TWAs:
  \[ O(|A|^2|t|) \]
  (Compute, for each node \( v \), the aggregated behavior of \( A \) on its subtree: \text{Behavior function})

- Nondeterministic (parallel) tree automata:
  \[ O(|A|^3|t|) \]
  (Compute, for each node \( v \), the set of reachable states)

- Nondeterministic TWAs:
  \[ O(|A|^3|t|) \]
  (Compute, for each node \( v \), the aggregated behavior of \( A \) on its subtree: \text{Behavior relation})
Facts

Time Bounds for the combined complexity of membership test for tree automata:

- Deterministic (parallel) tree automata: $O(|A||t|)$
- Nondeterministic (parallel) tree automata: $O(|A|^2|t|)$  
  (Compute, for each node, the set of reachable states)
- Deterministic TWAs: $O(|A|^2|t|)$  
  (Compute, for each node $v$, the aggregated behavior of $A$ on its subtree: Behavior function)

→ Nondeterministic TWAs: $O(|A|^3|t|)$  
  (Compute, for each node $v$, the aggregated behavior of $A$ on its subtree: Behavior relation)
Question: What is the structural complexity for the various models?

(Lohrey 2001, Segoufin 2003)

<table>
<thead>
<tr>
<th>Model</th>
<th>Time Complexity</th>
<th>Structural Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Det. top-down TA</td>
<td>$O(</td>
<td>\mathcal{A}</td>
</tr>
<tr>
<td>Det. bottom-up TA</td>
<td>$O(</td>
<td>\mathcal{A}</td>
</tr>
<tr>
<td>Nondet. bottom-up TA</td>
<td>$O(</td>
<td>\mathcal{A}</td>
</tr>
<tr>
<td>Nondet. top-down TA</td>
<td>$O(</td>
<td>\mathcal{A}</td>
</tr>
<tr>
<td>Det. TWA</td>
<td>$O(</td>
<td>\mathcal{A}</td>
</tr>
<tr>
<td>Nondet. TWA</td>
<td>$O(</td>
<td>\mathcal{A}</td>
</tr>
</tbody>
</table>
Non-emptiness for string automata corresponds to Graph Reachability (complete for NLOGSPACE)

Non-emptiness for tree automata corresponds to Path Systems
Facts

- Non-emptiness for string automata corresponds to Graph Reachability (complete for \textbf{NLOGSPACE})

- Non-emptiness for tree automata corresponds to \textbf{Path Systems}:

\[
\begin{array}{c}
p_1 \xrightarrow{} q \\ p_2 \xrightarrow{} q \end{array}
\]
Non-emptiness for string automata corresponds to Graph Reachability (complete for \textbf{NLOGSPACE})

Non-emptiness for tree automata corresponds to \textbf{Path Systems}:
Facts

- Non-emptiness for string automata corresponds to Graph Reachability (complete for \textbf{NLOGSPACE})

- Non-emptiness for tree automata corresponds to \textbf{Path Systems}:

\[
\begin{array}{c}
p_1 \\
\rightarrow \\
q \\
\downarrow \\
p_2
\end{array}
\]
Facts

- Non-emptiness for string automata corresponds to Graph Reachability (complete for NLOGSPACE)
- Non-emptiness for tree automata corresponds to Path Systems:

Result

- Non-emptiness for bottom-up tree automata can be checked in linear time
- It is complete for PTIME
Of course:

\[ L(A_1) = L(A_2) \iff [L(A_1) \subseteq L(A_2) \text{ and } L(A_2) \subseteq L(A_1)] \]

- Complexity of containment problem is very different for deterministic and non-deterministic automata
- Deterministic automata: construct product automaton
Deterministic bottom-up tree automata

- Product automaton analogous as for string automata
  - Set of states: $Q_1 \times Q_2$
  - Transitions component-wise
- To check $L(A_1) \subseteq L(A_2)$:
  - Compute $B = A_1 \times A_2$
  - Accepting states: $F_1 \times (Q_2 - F_2)$
  - Check whether $L(B) = \emptyset$
  - If so, $L(A_1) \subseteq L(A_2)$ holds

Theorem

Complexity of Containment for deterministic bottom-up tree automata:

$O(|A_1| \times |A_2|)$
Non-deterministic automata

- Naive approach:
  - Make $A_2$ deterministic (size: $O(2^{|A_2|})$)
  - Construct product automaton

$\Rightarrow$ Exponential time
Non-deterministic automata

- Naive approach:
  - Make $A_2$ deterministic (size: $O(2^{|A_2|})$)
  - Construct product automaton

$\Rightarrow$ Exponential time

Unfortunately...
There is essentially no better way
Non-deterministic automata

- Naive approach:
  - Make $A_2$ deterministic (size: $O(2^{|A_2|})$)
  - Construct product automaton

$\Rightarrow$ Exponential time

Unfortunately...

There is essentially no better way

Theorem (Seidl 1990)

Containment for non-deterministic tree automata is complete for EXPTIME
Theorem
Nonemptiness for deterministic top-down automata $\mathcal{A}$ can be checked in polynomial time.

Proof Idea
Check for each state $p$ of $\mathcal{A}$ and each pair $(q, q')$ of the leaves automaton $\mathcal{B}$:
Is there is a tree $t$ such that $\mathcal{A}$ starts from state $p$ and obtains a leave string which brings $\mathcal{B}$ from $q$ to $q'$?
Pre-XML
Decision Problems
Det. Top-Down Automata: Containment

Theorem
Containment for deterministic top-down automata $A$ can be checked in polynomial time

Proof Idea
- Tree automata $A_1$, $A_2$ with leaves automata $B_1$, $B_2$
- Check
  - for each pair $(p_1, p_2)$ of states of $A_1$ and $A_2$ and
  - for each two pairs $(q_1, q'_1)$ and $(q_2, q'_2)$ of $B_1$ and $B_2$, resp.:
    Is there a tree $t$ such that for both $i = 1, i = 2$: $T_i$ starts from state $p_i$ and obtains a leave string which brings $B_i$ from $q_i$ to $q'_i$?
### Complexities of basic algorithmic problems

<table>
<thead>
<tr>
<th>Model</th>
<th>Membership</th>
<th>Non-emptiness</th>
<th>Containment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Det. top-down TA</td>
<td>LOGSPACE</td>
<td>PTIME</td>
<td>PTIME</td>
</tr>
<tr>
<td>Det. bottom-up TA</td>
<td>LOGDCFL</td>
<td>PTIME</td>
<td>PTIME</td>
</tr>
<tr>
<td>Nondet. bottom-up TA</td>
<td>LOGCFL</td>
<td>PTIME</td>
<td>EXPTIME</td>
</tr>
<tr>
<td>Nondet. top-down TA</td>
<td>LOGCFL</td>
<td>PTIME</td>
<td>EXPTIME</td>
</tr>
<tr>
<td>Det. TWA</td>
<td>LOGSPACE</td>
<td>PTIME (*)</td>
<td>PTIME (*)</td>
</tr>
<tr>
<td>Nondet. TWA</td>
<td>NLOGSPACE</td>
<td>PTIME (*)</td>
<td>EXPTIME (*)</td>
</tr>
</tbody>
</table>

(∗: upper bounds)

---

**A further result to remember**

**Theorem (Stockmeyer, Meyer 1971)** Containment and Equivalence for regular expressions on strings are complete for **PSPACE**
Composer

Name
Claude Debussy

Vita
Born
When
1862
Where
Paris

Married
When
1899
Whom
Rosalie

Married
When
1908
Whom
Emma

Died
When
1918
Where
Paris

Piece
PTitle
La Mer

PYear
1905

Instruments
Large orchestra

Movements
3
Now we move from ranked to unranked trees

There is a basic choice:
  – Either: we encode unranked trees as binary trees and go on with ranked automata
  – Or: we adapt the ranked automata models

In both cases: not many surprises, most results remain
Encoding via ...

<table>
<thead>
<tr>
<th>first child</th>
<th>next sibling</th>
</tr>
</thead>
</table>

Example: Unranked Tree

```
      a
     /|
    c e
   /  |
  a   c
data  c  e
     /  |
    a   e
```
Encoding via ...

<table>
<thead>
<tr>
<th>first child</th>
</tr>
</thead>
<tbody>
<tr>
<td>next sibling</td>
</tr>
</tbody>
</table>

Example: Unranked Tree

```
  a
 / 
c   e
 /   |
a   c
 /   |
 a   e
```

Encoding as Binary Tree

```
  c
 / 
 a   e
 /   |
 a   a
 /   |
 a   e
```

PODS 2004 Thomas Schwentick Trees, Automata & XML
Ranked trees

Transitions are described by finite sets:
$$\delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\}$$
Transitions are described by finite sets:
\[ \delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\} \]
Transitions are described by finite sets:
\[ \delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\} \]
Ranked trees

Transitions are described by finite sets:
\[ \delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots \} \]

Unranked trees

\[ \delta(\sigma, q) \]

- For unranked trees, \( \delta(\sigma, q) \) is a regular language
- \( \delta(\sigma, q) \) can be specified by regular expression or finite string automaton

[Brüggemann-Klein, Murata, Wood 2001]
Remark

- Representation of $\delta(\sigma, q)$ has influence on complexity

- Natural choice:
  - For nondeterministic tree automata:
    represent $\delta(\sigma, q)$ by NFAs or regular expressions
  - For deterministic tree automata:
    represent $\delta(\sigma, q)$ by DFAs

$\Rightarrow$ Same complexity results as for ranked trees
The following are equivalent for a set $L$ of unranked trees:

(a) $L$ is accepted by a nondeterministic bottom-up automaton
(b) $L$ is accepted by a deterministic bottom-up automaton
(c) $L$ is accepted by a nondeterministic top-down automaton
(d) $L$ is characterized by an MSO-formula
| State at $v$ might depend on ... | ![Diagram](image)
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>state and symbol of parent</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>state and symbol of parent and symbol of $v$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>state and symbol of parent and symbols at $v$ and its left siblings</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>state and symbol of parent and symbols at $v$ and its siblings</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Fact
A simple deterministic top-down automata can check the existence of vertical paths with regular properties.

Construction
- For a node \( v \) let \( s(v) \) denote the sequence of labels from the root to \( v \)
- Let \( A \) be a deterministic string automaton
- \( A' := \) top-down automaton which takes at \( v \) state of \( A \) after reading \( s(v) \)

\[ \Rightarrow A' \text{ is deterministic} \]
- There is a path from the root to a leaf \( v \) with \( s(v) \in L(A) \) iff \( A' \) assumes at least one state from \( F \) at a leave

Streaming XML
Similar construction used for XPath evaluation on streams [Green et al. 2003]
Generalization of Tree-Walk Automata

Allowed transitions: 
- Go up
- Go to first child
- Go to left sibling
- Go to right sibling

→ Caterpillar automata [Brüggemann-Klein, Wood 2000]

Basic design choice

Should a transition to a sibling be aware of the label of the parent?

```
    a
   / \  \
  /   \ /  \
 /     /   /  \
v   w v   w
```
A third kind of automata for XML

- **Document automata** are string automata reading XML documents as text
- Tags are represented by symbols from a given alphabet
- Variants:
  - Finite document automata
  - Pushdown document automata
- Useful especially in the context of streaming XML

**Theorem (Segoufin, Vianu 2002)**

- Regular languages of XML-trees can be recognized by deterministic push-down document automata.
- Depth of push-down is bounded by depth of tree
Summary

- Moving from ranked to unranked automata requires some adaptations.
- Transitions can be defined with regular string languages $\delta(\sigma, q)$.
- By and large, things work smoothly.
- In particular:
  - There is an equally robust notion of regular tree languages.
  - The complexities are the same as for ranked automata (if the sets $\delta(\sigma, q)$ are represented in a sensible way).
XML-Automata

Document Automata

Refined Overview of Models

Non-det. top-down tree automata
Non-det. bottom-up tree automata
det. bottom-up tree automata
Pushdown document automata

Det. top-down tree automata

Non-det. tree walk automata

Det. tree walk automata

Finite document automata
Schemas  
Specialized DTDs  
Validation wrt a DTD

Example Tree

Composer
  Name  Vita
  Debussy  Born  Married  Married  Died
  When  Where  When  Whom  When  Whom  When  Where
  1862  Paris  1899  Rosalie  1908  Emma  1918  Paris

Example DTD

```xml
<!DOCTYPE Composers [  
  <!ELEMENT Composers (Composer*)>  
  <!ELEMENT Composer (Name, Vita, Piece*)>  
  <!ELEMENT Vita (Born, Married*, Died?)>  
  <!ELEMENT Born (When, Where)>  
  <!ELEMENT Married (When, Whom)>  
  <!ELEMENT Died (When, Where)>  
  <!ELEMENT Piece (PTitle, PYear,  
    Instruments, Movements)>  
]>
```

Validation Algorithm

For each node:
Check that the children are ok wrt the parent's rule
Observation

- Validation wrt DTDs is a simple task
- Can be done by
  - Bottom-up automata
  - Deterministic top-down automata
    (if siblings contribute to new state)
  - Deterministic tree-walk automata:
    Just a depth-first left-to-right traversal
- In particular: Validation possible in linear time during one pass through the document (1-pass validation)
- But DTDs are also rather weak...
A classical example

```xml
<!DOCTYPE Dealer [ 
  <!ELEMENT Dealer (UsedCars NewCars)>
  <!ELEMENT UsedCars (ad*)>
  <!ELEMENT NewCars (ad*)>
  <!ELEMENT ad ((model, year) | model)> ]>
```

Intention

Intention:

```
  Dealer
    UsedCars
      ad
      model
      year
    NewCars
      ad
      model
```

Observation

- Elements with the same name may have different structure in different contexts

→ It would be nice to have types for elements

→ Specialized DTDs
Definition (Papakonstantinou, Vianu 2000)

A specialized DTD (SDTD) over alphabet $\Sigma$ is a pair $(d, \mu)$, where

- $d$ is a DTD over the alphabet $\Sigma'$ of types
- $\mu : \Sigma' \rightarrow \Sigma$ maps types to tag names

Note

Concerning the name:
“specialized” refers to types, not to DTDs

Example

Dealer $\rightarrow$ UsedCars NewCars $\quad \mu(\text{Dealer}) = \text{Dealer}$
UsedCars $\rightarrow$ adUsed* $\quad \mu(\text{UsedCars}) = \text{UsedCars}$
NewCars $\rightarrow$ adNew* $\quad \mu(\text{NewCars}) = \text{NewCars}$
adUsed $\rightarrow$ model year $\quad \mu(\text{adUsed}) = \text{ad}$
adNew $\rightarrow$ model $\quad \mu(\text{adNew}) = \text{ad}$
### Example: SDTD for Boolean circuit trees

<table>
<thead>
<tr>
<th>Tag</th>
<th>$h(\text{Tag})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-AND</td>
<td>AND</td>
</tr>
<tr>
<td>1-OR</td>
<td>AND</td>
</tr>
<tr>
<td>0-AND</td>
<td>AND</td>
</tr>
<tr>
<td>1-leaf</td>
<td>1</td>
</tr>
<tr>
<td>0-leaf</td>
<td>0</td>
</tr>
</tbody>
</table>

1-AND $\rightarrow (1\text{-OR} \mid 1\text{-AND} \mid 1\text{-leaf})*$

1-OR $\rightarrow .\ast (1\text{-OR} \mid 1\text{-AND} \mid 1\text{-leaf}) .\ast$

0-AND $\rightarrow .\ast (0\text{-OR} \mid 0\text{-AND} \mid 0\text{-leaf}) .\ast$

0-OR $\rightarrow (0\text{-OR} \mid 0\text{-AND} \mid 0\text{-leaf})*$

1-leaf $\rightarrow \epsilon$

0-leaf $\rightarrow \epsilon$
A tree conforms to a specialized DTD \((d, \mu)\) if there is a labeling of its nodes by types which is valid wrt. \(d\).
Observation

- A tree conforms to a specialized DTD \((d, \mu)\) if there is a labeling of its nodes by types which is valid wrt. \(d\).
- This reminds us of something...
Observation

- A tree conforms to a specialized DTD \((d, \mu)\) if there is a labeling of its nodes by types which is valid wrt. \(d\)
- This reminds us of something...

Theorem

Specialized DTDs capture exactly the regular tree languages
Observation

- A tree conforms to a specialized DTD \((d, \mu)\) if there is a labeling of its nodes by types which is valid wrt. \(d\).
- This reminds us of something...

Theorem

Specialized DTDs capture exactly the regular tree languages.

Question: What about 1-pass validation?
**Definition (Validation)**
Given: Specialized DTD $d$, tree $t$
Question: Is $t$ valid wrt $d$?

**Definition (Typing)**
Given: Specialized DTD $d$, tree $t$
Output: Consistent type assignment for the nodes of $t$

**Facts**
- Specialized DTDs $\equiv$ regular tree languages
- Validation by a deterministic push-down automaton
- Validation in linear time during one pass through the document

**Question:** What about 1-pass typing?
Observations

- Type of a node $\equiv$ state of deterministic bottom-up automaton
- Deterministic push-down automaton can assign types during 1 pass
- But the type of a node $v$ is determined after visiting its subtree

...after visiting subtree
Observations

- Type of a node $\equiv$ state of deterministic bottom-up automaton
- Deterministic push-down automaton can assign types during 1 pass
- But the type of a node $v$ is determined after visiting its subtree
- 1-pass preorder typing:
  determine type of $v$ before visiting the subtree of $v$
I
Introduction
Pre-XML Tree Automata
Tree Automata for XML

II
Schemas
Node Selecting Queries
XPath Query Containment

III
Extensions
Conclusion

Specialized DTDs
1-pass Preorder Typing
Schema Containment
Question

When would it be important to know the type of $v$ before visiting the subtree of $v$?
Question
When would it be important to know the type of $v$ before visiting the subtree of $v$?

Answer
Whenever the processing proceeds in document order, e.g.:

- Streaming XML: Typing as the first operator in a pipeline
- SAX-based processing
Question
When would it be important to know the type of \( v \) before visiting the subtree of \( v \)?

Answer
Whenever the processing proceeds in document order, e.g.:
- Streaming XML: Typing as the first operator in a pipeline
- SAX-based processing

Our next goal
Find out which schemas admit 1-pass preorder typing
Restricted Schemas

(Murata, Lee, Mani 2001) introduced restrictions on specialized DTDs to ensure efficient validation

(∗: in a slightly different framework)

- Two types \( b, b' \) **compete** if \( \mu(b) = \mu(b') \)
- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., \( a \rightarrow bcb' \) is not single-type)
- A specialized DTD is **restrained-competition** if no rule allows strings \( wbv, wb'v' \) with competing types \( b, b' \)
  (e.g., \( a \rightarrow c(b + d^*b') \) is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs
(Murata, Lee, Mani 2001) introduced restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

Two types $b, b'$ compete if $\mu(b) = \mu(b')$

- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)

- A specialized DTD is **restrained-competition** if no rule allows strings $wbv$, $wb'v'$ with competing types $b, b'$ (e.g., $a \rightarrow c(b + d*b')$ is not restrained-competition)

- The authors argue that XML-Schema roughly corresponds to single-type SDTDs
(Murata, Lee, Mani 2001) introduced\(^*\) restrictions on specialized DTDs to ensure efficient validation (\(*\) in a slightly different framework)

- Two types \(b, b'\) **compete** if \(\mu(b) = \mu(b')\)

→ A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., \(a \rightarrow bcb'\) is not single-type)

- A specialized DTD is **restrained-competition** if no rule allows strings \(wbv\), \(wb'v'\) with competing types \(b, b'\)
  
  (e.g., \(a \rightarrow c(b + d^*b')\) is not restrained-competition)

- The authors argue that XML-Schema roughly corresponds to single-type SDTDs
Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types $b, b'$ **compete** if $\mu(b) = \mu(b')$

- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)

- A specialized DTD is **restrained-competition** if no rule allows strings $wbv$, $wb'v'$ with competing types $b, b'$ (e.g., $a \rightarrow c(b + d*b')$ is not restrained-competition)

- The authors argue that XML-Schema roughly corresponds to single-type SDTDs
Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation

(*: in a slightly different framework)

- Two types $b, b'$ compete if $\mu(b) = \mu(b')$

- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)

- A specialized DTD is **restrained-competition** if no rule allows strings $wbv$, $wb'v'$ with competing types $b, b'$

  (e.g., $a \rightarrow c(b + d^*b')$ is not restrained-competition)

→ The authors argue that XML-Schema roughly corresponds to single-type SDTDs
**Restricted Schemas**

(Murata, Lee, Mani 2001) introduced restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types $b, b'$ **compete** if $\mu(b) = \mu(b')$
- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)
- A specialized DTD is **restrained-competition** if no rule allows strings $wbv$, $wb'v'$ with competing types $b, b'$ (e.g., $a \rightarrow c(b + d*b')$ is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

**Fact**

Both restrictions are sufficient to get 1-pass preorder typing!
(Murata, Lee, Mani 2001) introduced restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types $b, b'$ compete if $\mu(b) = \mu(b')$
- A specialized DTD is single-type if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)
- A specialized DTD is restrained-competition if no rule allows strings $wbv$, $wb'v'$ with competing types $b, b'$ (e.g., $a \rightarrow c(b + d*b')$ is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Fact
Both restrictions are sufficient to get 1-pass preorder typing!

Question: Are they also necessary?
### Remarks

- The definition of “1-pass preorder typing” does not yet restrict the efficiency of determining the type of a node
- Typing could be 1-pass preorder but very time consuming
- It turns out that essentially this never happens
Remarks

- The definition of “1-pass preorder typing” does not yet restrict the efficiency of determining the type of a node
- Typing could be 1-pass preorder but very time consuming
- It turns out that essentially this never happens

Theorem (Martens, Neven, Sch. 2004)

For a regular tree language $L$ the following are equivalent

(a) $L$ can be described by a 1-pass preorder typable SDTD
(b) $L$ can be described by a restrained-competition SDTD
(c) $L$ has linear time 1-pass pre-order typing
(d) $L$ can be preorder-typed by a deterministic pushdown document automaton
(e) Types for trees in $L$ can be computed by a left-siblings-aware top-down deterministic tree automaton
Further characterizations
- This class has further interesting characterizations
- E.g., by closure under ancestor-sibling-guarded subtree exchange
Theorem (Martens, Neven, Sch. 2004)

For a regular tree language $L$ the following are equivalent

(a) $L$ can be described by a single-type SDTD

(b) Types for trees in $L$ can be computed by a simple top-down deterministic tree automaton

(c) $L$ is closed under ancestor-guarded subtree exchange
Introduction

Pre-XML Tree Automata

Tree Automata for XML

Schemas

Specialized DTDs

1-pass Preorder Typing

Schema Containment

Node Selecting Queries

XPath Query Containment

Extensions

Conclusion
### Schema Containment

**Given:** Schemas $d_1, d_2$

**Question:** Is $L(d_1) \subseteq L(d_2)$?

### Observations

- Important, e.g., for data integration
- Recall: Specialized DTDs are essentially non-deterministic tree automata

⇒ Containment of specialized DTDs is in **EXPTIME**

- But the restricted forms have lower complexity
- Complexity of containment depends on the allowed regular expressions
### Results (partly from Martens, Neven, Sch. 2004)

<table>
<thead>
<tr>
<th>Schema type</th>
<th>unrestricted</th>
<th>deterministic expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTDs</td>
<td>PSPACE</td>
<td>PTIME</td>
</tr>
<tr>
<td>single-type SDTDs</td>
<td>PSPACE</td>
<td>PTIME</td>
</tr>
<tr>
<td>restrained-competition SDTDs</td>
<td>PSPACE</td>
<td>PTIME</td>
</tr>
<tr>
<td>SDTDs</td>
<td>EXPTIME</td>
<td>EXPTIME</td>
</tr>
<tr>
<td>unrestricted SDTDs</td>
<td>EXPTIME</td>
<td>EXPTIME</td>
</tr>
</tbody>
</table>

### Observations
- For unrestricted SDTDs the complexity is dominated by tree automata containment.
- For the others it is dominated by the sub-task of checking containment for regular expressions.
Observations (cont.)

- ... for the others it is dominated by the sub-task of checking containment for regular expressions
- Actually, this observation can be made more precise

Theorem (Martens, Neven, Sch. 2004)

For a class $\mathcal{R}$ of regular expressions and a complexity class $\mathcal{C}$, the following are equivalent

(a) The containment problem for $\mathcal{R}$ expressions is in $\mathcal{C}$.

(b) The containment problem for DTDs with regular expressions from $\mathcal{R}$ is in $\mathcal{C}$.

(c) The containment problem for single-type SDTDs with regular expressions from $\mathcal{R}$ is in $\mathcal{C}$.
Regular tree languages are a nice framework for schema languages
  - Linear time validation
  - Static analysis is expensive
They also serve as a basis for restricted classes with better algorithmic properties:
  - 1-pass preorder typing
  - more feasible static analysis, in particular if the \( \delta(\sigma, q) \) are given by deterministic automata
Restrained competition \( \equiv \) Deterministic top-down automata \( \equiv \) 1-pass preorder typable
I
- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II
- Schemas
- Node Selecting Queries
- XPath Query Containment

III
- Extensions
- Conclusion
Introduction

Pre-XML Tree Automata

Tree Automata for XML

Schemas

Node Selecting Queries

XPath Query Containment

Extensions

Conclusion

A Robust Class

XPath and Pebble Automata
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born><When>August 22, 1862</When><Where>Paris</Where></Born>
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>
    <Died><When>March 25, 1918</When><Where>Paris</Where></Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```

Example query

```
//Vita/Died/*
```
Node-Selecting Q. A Robust Class  Node-Selecting Queries

Example document

Example query
//Vita/Died/*

Observation

XPath expressions define sets of nodes → node-selecting queries
Is there a class of node-selecting queries, as robust as the regular tree languages?

There is a simple way to define node selecting queries by monadic second-order formulas:

- Simply use one free variable: $\varphi(x)$
- Is there a corresponding automaton model?
- It is relatively easy to add node selection to nondeterministic bottom-up automata

**Definition (Nondeterministic bottom-up node-selecting automata)**

- Nondeterministic bottom-up automata plus select function:
  $$s : Q \times \Sigma \rightarrow \{0, 1\}$$

- Node $v$ is in result set for tree $t$ if there is an accepting computation on $t$ in which $v$ gets a state $q$ such that $s(q, \lambda(v)) = 1$
Example query

///*[a]///b

Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: $q_0$
Example query

```c
// *[a] //b
```

**Example automaton**

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) = \epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: $q_0$

**Example automaton**

```
Example tree: Run 1
```

**Query tree**
Example query
/**/[a]**//b

Example automaton
- \( Q = \{q_0, q_a, q_b\} \)
- \( L(q_a, a) = Q^* \)
- \( L(q_b, \sigma) = Q^* \)
- \( L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^* \)
- all other sets empty
- \( s(q_b, b) = 1 \)
- all others: 0
- Accepting: \( q_0 \)
Example query
//*[a]//b

Example automaton
- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: $q_0$
**Example query**
\\texttt{///*[a]/\b}

**Example automaton**

- \( Q = \{q_0, q_a, q_b\} \)
- \( L(q_a, a) = Q^* \)
- \( L(q_b, \sigma) = Q^* \)
- \( L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^* \)
- all other sets empty
- \( s(q_b, b) = 1 \)
- all others: 0
- Accepting: \( q_0 \)

**Example tree: Run 1**

**Query tree**

```
  *  
  / \   
 a   b
```

Pods 2004 Thomas Schwentick
Trees, Automata & XML 88
Example query

///*[a]///b

Example automaton

- \( Q = \{ q_0, q_a, q_b \} \)
- \( L(q_a, a) = Q^* \)
- \( L(q_b, \sigma) = Q^* \)
- \( L(q_0, \sigma) = \epsilon + q_0^* + Q^* q_a Q^* \)
- all other sets empty
- \( s(q_b, b) = 1 \)
- all others: 0
- Accepting: \( q_0 \)
Example query

//*[a]//b

Example automaton

- \( Q = \{ q_0, q_a, q_b \} \)
- \( L(q_a, a) = Q^* \)
- \( L(q_b, \sigma) = Q^* \)
- \( L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^* \)
- all other sets empty
- \( s(q_b, b) = 1 \)
- all others: 0
- Accepting: \( q_0 \)
Example query

`//*[a]//b`

Example automaton

- $Q = \{ q_0, q_a, q_b \}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: $q_0$
Example query
//*[a]//b

Example automaton
- \( Q = \{ q_0, q_a, q_b \} \)
- \( L(q_a, a) = Q^* \)
- \( L(q_b, \sigma) = Q^* \)
- \( L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^* \)
- all other sets empty
- \( s(q_b, b) = 1 \)
- all others: 0
- Accepting: \( q_0 \)

Query tree

Example tree: Run 2
Example query
///*[a]///b

Example automaton
- \( Q = \{ q_0, q_a, q_b \} \)
- \( L(q_a, a) = Q^* \)
- \( L(q_b, \sigma) = Q^* \)
- \( L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^* \)
- all other sets empty
- \( s(q_b, b) = 1 \)
- all others: 0
- Accepting: \( q_0 \)

Query tree

Example tree: Run 2

\[ Q = \{ q_0, q_a, q_b \} \]
\[ L(q_a, a) = Q^* \]
\[ L(q_b, \sigma) = Q^* \]
\[ L(q_0, \sigma) = \epsilon + q_0^* + Q^*q_aQ^* \]
all other sets empty
\[ s(q_b, b) = 1 \]
all others: 0
Accepting: \( q_0 \)
Fact

- Existential semantics: a node is in the result if there exists an accepting run which selects it
- Universal semantics: a node is in the result if every accepting run selects it
- Both semantics define the same class of queries

Result

A node selecting query is MSO-definable iff it is expressible by a nondeterministic bottom-up node selecting automaton
Result
A node selecting query is MSO-definable iff it is expressible by a nondeterministic bottom-up node selecting automaton

Proof Idea
- Given formula $\varphi(x)$ of quantifier-depth $k$ and tree $t$,
  for each node $v$ the automaton does the following:
  - Compute $k$-type of subtree at $v$
  - Guess $k$-type of ”envelope tree” at $v$
  - Conclude whether $v$ is in the output
  - Check consistency upwards towards the root

$\Rightarrow$ one unique accepting run

Crucial fact
\[ e_1 \equiv_k t_1 \quad e_2 \equiv_k t_2 \quad \Rightarrow \quad e_1 \equiv_k t_1 \quad e_2 \equiv_k t_2 \]
Node-Selecting Q.  A Robust Class  Equivalent Models

**More query models**

- Unfortunately, the translation from formula to automaton can be prohibitively expensive: number of states \( \sim 2^{2 \cdot 2^{2^{|\varphi|}}} \)

- Actually: If \( P \neq NP \) there is no elementary \( f \), such that MSO-formulas can be evaluated in time \( f(|\text{formula}| \times p(|\text{tree}|)) \) with polynomial \( p \) [Frick, Grohe 2002]

  → query languages with better complexity properties needed

- Good candidate: Monadic Datalog [Gottlob, Koch 2002] and its restricted dialects like TMNF

- Further models:
  - Attributed Grammars [Neven, Van den Bussche 1998]
  - \( \mu \)-formulas [Neumann 1998]
  - Context Grammars [Neumann 1999]
  - Deterministic Node-Selecting Automata [Neven, Sch. 1999]
Some facts about query evaluation

• MSO node-selecting queries can be evaluated in two passes through the tree
  – first pass, bottom-up: essentially computes the types of the subtrees
  – second pass, top-down: essentially computes the types of the envelopes and combines it with the subtree information

• This can be implemented by a 2-pass pushdown document automaton which in its first pass attaches information to each node

  [Neumann, Seidl 1998; Koch 2003]

• In particular: queries can be evaluated in linear time
Node-Selecting Q. XPath Pebble Automata: Example

Example Query Tree

Example Tree
Node-Selecting Q. XPath Pebble Automata: Example

Example Query Tree

Example Tree
Example Query Tree

Example Tree
Node-Selecting Q.  XPath  Pebble Automata: Example

Example Query Tree

Example Tree
Node-Selecting Q.  XPath  Pebble Automata: Example

Example Query Tree

Example Tree

PODS 2004  Thomas Schwentick  Trees, Automata & XML
Node-Selecting Q.  XPath  Pebble Automata: Example

Example Query Tree

Example Tree

PODS 2004  Thomas Schwentick  Trees, Automata & XML  94
Example Query Tree

```
     b
    / \
   a   e
   |   |
  b  d   f
 |   |   
c   g
```

Example Tree

```
  b
 / \
a   1
   / \
c   d
  /   
  e   
  /     
 b     
    /   
   c     
    /     
   f     
    /     
   c     
    /     
   g
```

Node-Selecting Q. XPath Pebble Automata: Example
Example Query Tree

Example Tree

PODS 2004
Thomas Schwentick
Trees, Automata & XML
Node-Selecting Q.  XPath  Pebble Automata: Example

Example Query Tree

Example Tree
Example Query Tree

Example Tree

Node-Selecting Q.  XPath  Pebble Automata: Example
Node-Selecting Q. XPath Pebble Automata: Example

Example Query Tree

Example Tree
Node-Selecting Q. XPath Pebble Automata: Example

Example Query Tree

Example Tree

PODS 2004 Thomas Schwentick Trees, Automata & XML 94
Example Query Tree

Example Tree
Definition (Pebble Automata)

- Extension of tree-walk automata by fixed number $k$ of pebbles
- Only pebble with highest number (current pebble) can move, depending on state, number of pebbles symbols under pebbles and incidence of pebbles
- Possible pebble movements:
  - stay, go to left sibling, go to right sibling, go to parent
  - lift current pebble or place new pebble at current position
- Nondeterminism possible

Facts

- Pebble automata capture navigational XPath queries
- Extended by alternation, branching and an output mechanism they even capture a large part of XSLT

[Papakonstantinou, Vianu 2000]
Some observations

- On strings, MSO logic and (unary) transitive closure logic (TC-logic) coincide
- On trees
  - MSO \equiv parallel automata
  - TC-logic \equiv pebble automata (i.e., strongest sequential automata)
- Whether MSO \equiv TC-logic is open
- First-order logic \equiv XPath + conditional axes [Marx 2004]
- The relationship between logics and automata models between FO and TC-logic is largely unexplored:
  - Tree-walk automata,
  - FO-logic + regular expressions
  - Conditional XPath + arbitrary star operator
  - ...
There is a natural notion of regular node-selecting queries generalizing regular tree languages.

- Probably for most practical purposes too strong.
- But it offers a useful framework for the study of other classes of queries.
- A robust but weaker class of queries is captured by pebble automata.
I
Introduction

Pre-XML Tree Automata

Tree Automata for XML

II
Schemas

Node Selecting Queries

XPath Query Containment

III
Extensions

Conclusion
Example document

```
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born><When>August 22, 1862</When><Where>Paris</Where></Born>
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>
    <Died><When>March 25, 1918</When><Where>Paris</Where></Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born><When>August 22, 1862</When><Where>Paris</Where></Born>
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>
    <Died><When>March 25, 1918</When><Where>Paris</Where></Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```

Example query
```
//Vita/Died/*
```
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born><When>August 22, 1862</When><Where>Paris</Where></Born>
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>
    <Died><When>March 25, 1918</When><Where>Paris</Where></Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```
Example document:

```
<Composer>
  <Name> Claude Debussy </Name>
  <Vita>
    <Born> <When> August 22, 1862 </When> <Where> Paris </Where> </Born>
    <Married> <When> October 1899 </When> <Whom> Rosalie </Whom> </Married>
    <Married> <When> January 1908 </When> <Whom> Emma </Whom> </Married>
    <Died> <When> March 25, 1918 </When> <Where> Paris </Where> </Died>
  </Vita>
  <Piece>
    <PTitle> La Mer </PTitle>
    <PYear> 1905 </PYear>
    <Instruments> Large orchestra </Instruments>
    <Movements> 3 </Movements>
    ...
  </Piece>
  ...
</Composer>
```

Another example query:

```
(/*[Name]//When) | (//Where)
```
Another example query

\((/*[\text{Name}]/\text{When}) \mid (//\text{Where})\)

Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born>
      <When>August 22, 1862</When>
      <Where>Paris</Where>
    </Born>
    <Married>
      <When>October 1899</When>
      <Whom>Rosalie</Whom>
    </Married>
    <Married>
      <When>January 1908</When>
      <Whom>Emma</Whom>
    </Married>
    <Died>
      <When>March 25, 1918</When>
      <Where>Paris</Where>
    </Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>

    ...
  </Piece>

  ...
</Composer>
```

More XPath operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p/q)</td>
<td>child</td>
</tr>
<tr>
<td>(p//q)</td>
<td>descendant</td>
</tr>
<tr>
<td>(p[q])</td>
<td>filter</td>
</tr>
<tr>
<td>(\ast)</td>
<td>wildcard</td>
</tr>
<tr>
<td>(p \mid q)</td>
<td>disjunction</td>
</tr>
</tbody>
</table>
Example document

```xml
<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born><When>August 22, 1862</When><Where>Paris</Where></Born>
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>
    <Died><When>March 25, 1918</When><Where>Paris</Where></Died>
  </Vita>
  <Piece>
    <PTitle>La Mer</PTitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
    ...
  </Piece>
  ...
</Composer>
```

Another example query

```
(//*[Name]//When) | (//'Where)
```

More XPath operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p/q$</td>
<td>child</td>
</tr>
<tr>
<td>$p//q$</td>
<td>descendant</td>
</tr>
<tr>
<td>$p[q]$</td>
<td>filter</td>
</tr>
<tr>
<td>$*$</td>
<td>wildcard</td>
</tr>
<tr>
<td>$p</td>
<td>q$</td>
</tr>
</tbody>
</table>
Question

Does 

\[
\text{/Vita/Died/}\star
\]

always select a subset of positions of 

\[
\text{(/*[Name]/When)} \mid (//Where)
\]
**Question**

Does `//Vita/Died/*` always select a subset of positions of `(/*[Name]//When) | (//Where)`?

**Answer**

No!
XPath Query Containment.

Question

Does \(/Vita/Died/*\) always select a subset of positions of \((/*[Name]//When) | (//Where)\)?

Answer

No!

Counter-example

\(<Vita>
  \(<Died>
    \(<How>Heart disease</How>
  </Died>
</Vita>\)
Question
Does \(//\text{Vita}/\text{Died}/*\) always select a subset of positions of \((/*[\text{Name}]/\text{When}) \mid (//\text{Where})\)?

Answer
No!

Counter-example
\[
\langle \text{Vita} \rangle
\langle \text{Died} \rangle
\langle \text{How} \rangle \text{ Heart disease } \langle /\text{How} \rangle
\langle /\text{Died} \rangle
\langle /\text{Vita} \rangle
\]

Further question
But what if the type of documents is constrained?
Fact

For all XML documents of type

```xml
<!DOCTYPE Composers [
  <!ELEMENT Composers (Composer*)>
  <!ELEMENT Composer (Name, Vita, Piece*)>
  <!ELEMENT Vita (Born, Married*, Died?)>
  <!ELEMENT Born (When, Where)>
  <!ELEMENT Married (When, Whom)>
  <!ELEMENT Died (When, Where)>
  <!ELEMENT Piece (PTitle, PYear, Instruments, Movements)>
]
```

the pattern `//Vita/Died/*` always selects a subset of positions of

```xml
(/*[Name]//When) | (//Where)
```
**XPath Query Containment.**

**XPath Containment: Definition**

**Definition (Containment for XPath(S))**
Let $S$ be a set of XPath-operators. The containment problem for $\text{XPath}(S)$ is:

**Given:** $\text{XPath}(S)$-expression $p, q$

**Question:** Is $p(t) \subseteq q(t)$ for all documents $t$?

**Definition (Containment for XPath (S) with DTD)**
Let $S$ be a set of XPath-operators. The containment problem for $\text{XPath}(S)$ in the presence of DTDs is:

**Given:** $\text{XPath}(S)$-expression $p, q$, DTD $d$

**Question:** Is $p(t) \subseteq q(t)$ for all documents $t$ satisfying $t \models d$?

**Observation**
These problems are crucial for static analysis and query optimization.

**Question**
For which fragments $S$ are these problems
- decidable?
- efficiently solvable?
General remarks

- The XPath containment problem has been considered for various sets of operators
- Results vary from PTIME to “undecidable”
- Various methods have been used:
  - Canonical model technique
  - Homomorphism technique
  - Chase technique
- More about this in [Miklau, Suciu 2002; Deutsch, Tanen 2001; Sch. 2004]
- We will consider automata based techniques
Definition (Relative Containment for XPath \((S')\) wrt DTD)

Let \(S\) be a set of XPath-operators. The containment problem for XPath\((S')\) relative to a DTD is:

**Given:** XPath\((S')\)-expression \(p, q, \text{DTD } d\)

**Question:** Is \(p(D) \subseteq q(D)\) for all documents \(D\) satisfying \(D \models d\)?

A vague plan

- Construct an automaton \(A_p\) for \(p\)
- Construct an automaton \(A_q\) for \(q\)
- Construct an automaton \(A_d\) for \(d\)
- Combine these automata suitably to get an automaton which accepts all counter-example documents
Definition (Boolean containment)

\[ p \subseteq_b q \iff \text{whenever } p \text{ selects some node in a tree } t \text{ then } q \text{ also selects some node in } t. \]

Useful observation [Miklau, Suciu 2002]
In the presence of \([\text{ ]}\), Boolean containment has the same complexity as containment.

Crucial idea

If and only if

\[ p_1 \neq p_2 \]

\[ p_1' \neq p_2' \]
The Boolean containment problem for $\text{XPath}(\text{/}, \text{//})$ in the presence of DTDs is in $\text{PTIME}$

The Boolean containment problem for $\text{XPath}(\text{/}, \text{//}, [], *, |)$ in the presence of DTDs is in $\text{EXPTIME}$

Note
Both results are optimal wrt complexity: the problems are complete for these classes
Result 1 [Neven, Sch. 2003]
The Boolean containment problem for XPath(/, //) in the presence of DTDs is in $\text{PTIME}$

Proof Idea

- XPath(/, //)-expressions can only describe vertical paths in a tree.
- Each expression is basically of the form $p_1//p_2//\cdots//p_k$, where each $p_i$ is of the form $l_{i1}//\cdots//l_{im_i}$.
- On strings this is a sequence of string matchings corresponding to a regular language $L$.
  $$\Rightarrow$$ Deterministic string automaton of linear size.
- Recall: there is a deterministic top-down automaton which checks whether a $p$-path exists.
  $$\Rightarrow$$ Deterministic top-down automaton $A_p$.
- $$\Rightarrow$$ Deterministic top-down automaton $A_q$ checking that no $q$-path exists.
Result 1 [Neven, Sch. 2003]
The containment problem for XPath(//, //) in the presence of DTDs is in PTIME

Proof idea (cont.)
- Deterministic top-down automaton $A_p$
- Deterministic top-down automaton $A_q$ checking that no $q$-path exists
- There is a deterministic top-down automaton $A_d$ checking whether $t$ conforms to $d$
- $p \subseteq_b q$ in the presence of $d \iff L(A_p \times A_q \times A_d) = \emptyset$
- The latter can be checked in polynomial time
**XPath Query Containment.**

**Containment for XPath(/, //, [], *, |) and DTDs**

**Result 2 [Neven, Sch. 2003]**

The containment problem for \( \text{XPath}(/\text{,} //, [], *, |) \) in the presence of DTDs is in \textbf{EXPTIME}

**Proof Idea**

We again represent patterns like

\[
/*[\text{Name}]//\text{When} | (//\text{Where})
\]

as query trees:

**Example query tree**

\[
\text{Name} \rightarrow * \rightarrow \text{Where} \rightarrow \text{When}
\]

**Lemma**

For each \( \text{XPath}(/\text{,} //, [], *, |) \)-expression \( p \) there is a deterministic bottom-up automaton \( \mathcal{A}_p \) of exponential size which checks whether in a tree \( p \) holds
Lemma

For each $\text{XPath}(/, //, [], *, |)$-expression $p$ there is a deterministic bottom-up automaton $A_p$ of exponential size which checks whether in a tree $p$ holds.

Proof idea for Lemma

- States of $A_p$ are of the form $(S_/, S_{//})$
- Both $S_/$ and $S_{//}$ are sets of positions of the query tree:
  - $S/$: positions matching $v$
  - $S_{//}$: positions matching some node in the subtree of $v$
Result 2 [Neven, Sch. 2003]
The containment problem for XPath(//, //, [], *, |) in the presence of DTDs is in \textbf{EXPTIME}

Proof idea (cont.)
- Construct deterministic bottom-up automaton $A_p$ of exponential size
- Construct deterministic bottom-up automaton $A_q$ of exponential size
- Construct deterministic bottom-up automaton $A_d$ of exponential size
- $p \subseteq_b q$ in the presence of $d \iff L(A_p \times A_q \times A_d) = \emptyset$
  $\Rightarrow$ exponential time
Summary (Automata and XPath containment)

- Automata are a useful algorithmic tool
- In particular, if several algorithmic tasks have to be combined
- Complexity depends on type of automata

Summary (XPath containment in general)

- Many more results in other papers, e.g., [Miklau, Suciu 2002; Deutsch, Tanen 2001; Sch. 2004]
- The complexity of XPath query containment varies strongly with the allowed operators
- Even undecidable in general
- Exact borderline between undecidable and decidable has to be identified
Pebble automata

- As mentioned before: XSLT transformations can be modeled by \( k \)-pebble transducers (\( k \)-pebble automata + alternation, branching, output)
- Pebbles are mainly used to evaluate XPath expressions

XSLT Typechecking problem

**Given:** Transformation \( T \), Schemas \( d_1, d_2 \)

**Question:** Is \( T(t) \) valid wrt \( d_2 \) whenever \( t \) is valid wrt \( d_1 \)?

Theorem (Milo, Suciu, Vianu 2000)
The typechecking problem for (core) XSLT is decidable
Theorem (Milo, Suciu, Vianu 2000)
The typechecking problem for (core) XSLT is decidable

Proof Idea

- Obvious approach:
  - Compute $T(L(d_1))$
  - Check that $T(L(d_1)) \subseteq L(d_2)$

- Problem: $T(L(d_1))$ does not need to be regular:
  Transform \[
  \begin{array}{c}
  b \\
  a \\
  a \\
  a \\
  a \\
  \end{array}
  \]
  into
  \[
  \begin{array}{c}
  b \\
  a \\
  a \\
  a \\
  a \\
  a \\
  \end{array}
  \]

- Better approach:
  Compute $T^{-1}(L(d_2))$ and check $L(d_1) \subseteq T^{-1}(L(d_2))$
Proof idea (cont.)

- **\( k \)-pebble acceptor**: \( k \)-pebble transducer without output

- Prove: \( T^{-1}(L) \) is accepted by a \( k \)-pebble acceptor if \( L \) is regular

- Prove: Behavior of \( k \)-pebble acceptors can be described by MSO-formulas

\[ \Rightarrow \] \( k \)-pebble acceptors only accept regular tree languages

\[ \Rightarrow \] \( T^{-1}(L(d_2)) \) is regular

- Algorithm:
  - Construct automaton for \( T^{-1}(L(d_2)) \)
  - Construct an equivalent MSO-formula \( \varphi \)
  - Construct bottom-up tree automaton \( A \) for \( \neg \varphi \)
  - Check that \( L(d_1) \subseteq L(A) \)

- Complexity: VERY bad (non-elementary)
So far...

- We have seen that automata are useful for
  - Validation, Typing
  - Navigation
  - Transformation

- What about more general queries?
  - results of higher arity?
  - joins, i.e., comparisons of data values
  - counting

- Are automata useful for XQuery?

- ... for tree pattern queries?
Higher arity

- Nonemptiness and containment questions can be handled by automata: tuples can be encoded by additional labels
- What about query evaluation for higher arity?

Data values

- When data values in XML documents are taken into account, things become more complicated, e.g.:
  - Even First-order logic becomes undecidable
  - Pebble automata become undecidable
  - Automata with data registers become undecidable when they are allowed to move up and down
- What is the right notion for regular (string) languages over infinite alphabets?
- What are sensible decidable restrictions of logics and automata in the context of data values?
Counting

- Automata can be equipped with counting facilities, e.g.:
  - Presburger tree automata: $\delta(\sigma, q)$ is Boolean combination of
    - regular expressions and
    - quantifier-free Presburger formulas like
      "number of children in state $q_1 = number of children in state q_2"

- Nondet. Presburger automata:
  - $\equiv$ MSO logic
  - Whether automaton accepts all trees is undecidable

- Det. Presburger automata:
  - $\equiv$ Presburger $\mu$-formulas
  - Membership test: $O(|A||t|)$
  - Non-emptiness: PSPACE
  - Containment: PSPACE

[Seidl, Sch., Muscholl, Habermehl 2004]
Introduction

Pre-XML Tree Automata

Tree Automata for XML

Schemas

Node Selecting Queries

XPath Query Containment

Extensions

Conclusion
We saw...

- A broad variety of automata models which can be used for XML and its theory
- Well-established in the context of validation, typing, navigation, transformation
- Well-established as
  - means to define robust classes
  - proof tools
  - algorithmic tools

Big question
Can automata be employed as a tool for XQuery evaluation?