Logic and XML

München
May 2005

Thomas Schwentick
## Contents

**Introduction**
- XML: Tasks
- Logic on Trees

**MSO Logics**

**Weaker Logics**

**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>
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
## 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

<table>
<thead>
<tr>
<th>Kind</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Validation</strong></td>
<td>DTD, XML Schema</td>
</tr>
<tr>
<td>Check whether an XML document is of a given type</td>
<td></td>
</tr>
<tr>
<td><strong>Navigation</strong></td>
<td>XPath</td>
</tr>
<tr>
<td>Select a set of positions in an XML document</td>
<td></td>
</tr>
<tr>
<td><strong>Querying</strong></td>
<td>XQuery</td>
</tr>
<tr>
<td>Extract information from an XML document</td>
<td></td>
</tr>
<tr>
<td><strong>Transformation</strong></td>
<td>XSLT</td>
</tr>
<tr>
<td>Construct a new XML document from a given one</td>
<td></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>
```

...
Validation: DTD

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</PTtitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</Movements>
  </Piece>
</Composer>
```

DTDs describe types of XML documents
**Validation: DTD**

**DTD**

DTDs describe types of XML documents

**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>  
      <When>August 22, 1862</When>  
      <Where>Paris</Where>  
    </Born>  
    <Married>  
      <When>October 1899</When>  
      <Whom>Rosalie</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>  
...
### Composer

**Name**: Claude Debussy

**Vita**

- **Born**: August 22, 1862, Paris
- **Married**: October 1899, Rosalie
- **Married**: January 1908, Emma
- **Died**: March 25, 1918, Paris

**Piece**

- **PTitle**: La Mer
- **PYear**: 1905
- **Instruments**: Large orchestra
- **Movements**: 3
XPath expressions select sets of nodes of XML documents by specifying navigational patterns.
Navigation: **XPath**

**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>
```

**XPath**

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

**Example query**

```
//Vita/Died/*
```
XPath expressions select sets of nodes of XML documents by specifying navigational patterns.

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

<Composer>
  <Name>Claude Debussy</Name>
  <Vita>
    <Born>
      <When>August 22, 1862</When>
      <Where>Paris</Where>
    </Born>
  </Vita>
  <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>
</Composer>

<Piece>
  <PTitle>La Mer</PTitle>
  <PYear>1905</PYear>
  <Instruments>Large orchestra</Instruments>
  <Movements>3</Movements>
</Piece>

XPath

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

Example query

//Vita/Died/*

Remark

XPath expressions define sets of nodes: node-selecting queries
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>
      <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 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

```
(Name) Claude Debussy (/Name)
(Name) Eric Satie (/Name)
(Name) Hector Berlioz (/Name)
(Name) Camille Saint-Saëns (/Name)
(Name) Frédéric Chopin (/Name)
(Name) Maurice Ravel (/Name)
(Name) Jim Morrison (/Name)
(Name) César Franck (/Name)
(Name) Gabriel Fauré (/Name)
(Name) George Bizet (/Name)
...
Transformation: XSLT

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</PTtitle>
    <PYear>1905</PYear>
    <Instruments>Large orchestra</Instruments>
    <Movements>3</ Movements>
  </Piece>
</Composer>
```
Transformation: XSLT

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>
```

XSLT

XSLT transforms documents by means of templates
**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>
```

**XSLT**

XSLT transforms documents by means of templates

**Example**

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

XSLT:

```xml
<xsl:template match="Composer[Vita//Where='Paris']">
  <ParisComposer>
    <Name>Claude Debussy</Name>
    <Born>
      <When>August 22, 1862</When>
      <Where>Paris</Where>
    </Born>
    <ParisComposer>
      <Name>Frédéric Chopin</Name>
      <Born>
        <When>March 1, 1810</When>
        <Where>Żelazowa</Where>
      </Born>
    </ParisComposer>
    <ParisComposer>
      <Name>Camille Saint-Saëns</Name>
      <Born>
        <When>October 9, 1835</When>
        <Where>Paris</Where>
      </Born>
    </ParisComposer>
  </ParisComposer>
</xsl:template>
```

Result:

```xml
<ParisComposer>
  <Name>Claude Debussy</Name>
  <Born>
    <When>August 22, 1862</When>
    <Where>Paris</Where>
  </Born>
  <ParisComposer>
    <Name>Frédéric Chopin</Name>
    <Born>
      <When>March 1, 1810</When>
      <Where>Ẓelazowa</Where>
    </Born>
  </ParisComposer>
  <ParisComposer>
    <Name>Camille Saint-Saëns</Name>
    <Born>
      <When>October 9, 1835</When>
      <Where>Paris</Where>
    </Born>
  </ParisComposer>
</ParisComposer>
```
A Schematic View

DTD/XML Schema

XPath

XQuery

XSLT

yes/no
XML Languages
The Big Picture

XML Languages

Known Formal Models
The Big Picture

XML Languages

Known Formal Models
The Big Picture

XML Languages

Known Formal Models

Suitable Fragments
## Algorithmic Tasks

### Evaluation

<table>
<thead>
<tr>
<th>Evaluation (Combined)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Tree ( t ), Query ( q )</td>
</tr>
<tr>
<td>O: ( q(t) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Evaluation (Data(( q )))</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Tree ( t )</td>
</tr>
<tr>
<td>O: ( q(t) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incremental Eval. (( q ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Tree ( t ), Changes of ( t )</td>
</tr>
<tr>
<td>O: ( q(t) )</td>
</tr>
</tbody>
</table>

### Static Analysis

<table>
<thead>
<tr>
<th>Satisfiability</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Query ( q )</td>
</tr>
<tr>
<td>Q: Is ( q(t) \neq \emptyset ) for some ( t )?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Containment</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Queries ( q_1, q_2 )</td>
</tr>
<tr>
<td>Q: Is always ( q_1(t) \subseteq q_2(t) )?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Queries ( q_1, q_2 )</td>
</tr>
<tr>
<td>Q: Is always ( q_1(t) = q_2(t) )?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Types ( d_1, d_2 ), Transformation ( T )</td>
</tr>
<tr>
<td>Q: Does ( t \models d_1 ) imply ( T(t) \models d_2 )?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Types ( d ), Transformation ( T )</td>
</tr>
<tr>
<td>O: Type of ( { T(t) \mid t \models d } )</td>
</tr>
</tbody>
</table>
A Look Back

Relational Databases

- SQL
- Relational Algebra
- FO-Logic

Basic Properties of these Formalisms

**SQL**
- Declarative, easy to use
- Queries, data definition, updates

**FO-logic**
- Formal framework for investigations
- Clear Semantics
- Expressive power well understood

**Relational Algebra**
- Operational model
- Flexible, optimizable
- Automatic translation
A Look Back

Relational Databases
- SQL
- Relational Algebra
- FO-Logic

Further Properties
- Satisfiability undecidable
  → Fragments like conjunctive queries
  → Evaluation for conjunctive queries NP-hard
  → but works well in practice
- SQL can count and group
  → Can be added to FO

Basic Properties of these Formalisms

SQL
- Declarative, easy to use
- Queries, data definition, updates

FO-logic
- Formal framework for investigations
- Clear Semantics
- Expressive power well understood

Relational Algebra
- Operational model
- Flexible, optimizable
- Automatic translation

[Hella et al. 01]
A Look Back

Relational Databases

SQL

Relational Algebra

FO-Logic

Basic Properties of these Formalisms

SQL
- Declarative, easy to use
- Queries, data definition, updates

FO-logic
- Formal framework for investigations
- Clear Semantics
- Expressive power well understood

Relational Algebra
- Operational model
- Flexible, optimizable
- Automatic translation

Further Properties

- Satisfiability undecidable
  \[ \xrightarrow{\text{Fragments like conjunctive queries}} \]
- Evaluation for conjunctive queries NP-hard
  \[ \xrightarrow{\text{but works well in practice}} \]
- SQL can count and group
  \[ \xrightarrow{\text{Can be added to FO}} \]

Goal

XML-Languages

Automata

??-Logic

[Hella et al. 01]
Contents

Introduction

XML: Tasks

Logic on Trees

MSO Logics

Weaker Logics

Extensions

Conclusion
Example Document

\<a\> \<b\>
  \<d\>12\</d\> \<e\>22\</e\>
\</b\>
\<b\> \<d\>4\</d\> \<b\>
\<b\> \<d\>11\</d\>
  \<e\>8\</e\> \<e\>18\</e\> \<e\>5\</e\>
\</b\>
\<c\> \<b\> \<f\>2\</f\>
\</b\>7\</c\>
\</a\>

...as Binary Tree

```
Tree:
```

...as Unranked Tree

```
Tree:
```

M"unchen May 05 Logic and XML 13 Thomas Schwentick
XML Trees as Finite Models

Data Values

For many investigations, data values can and have to be ignored

Example

```
  a
 / \  
 b   b  
 \   /  
 d e 18 
  12 22 4 11 8 5  
```

Labels

- Usually, XML trees are modeled as labeled trees over a finite alphabet
- For schema languages this is ok
- For queries, the actual alphabet might depend on the query
- Unary predicates on data values can be also modeled this way

Signatures

- There are various ways to represent unordered trees as finite relational models
- Possible relations
  - Local orders: \( \rightarrow, \downarrow \)
  - Their transitive closures: \( \rightarrow^+, \downarrow^+, \leftarrow^+, \uparrow^+ \)
  - Reflexive and transitive: \( \rightarrow^*, \downarrow^*, \leftarrow^*, \uparrow^* \)
  - Document order: \( \uparrow^*, \downarrow^* \)
- Frequent combinations:
  - \( \rightarrow, \downarrow \)
  - \( \rightarrow^+, \downarrow^+ \)
  - \( \rightarrow, \downarrow, \rightarrow^+, \downarrow^+ \)
  - \( \rightarrow, \downarrow \)
**XML Trees as Finite Models**

### Data Values
For many investigations, data values can and have to be ignored.

### Example
```
   a
  / \  
 b   c
 / \  
 d   e
    
 f
```

### Labels
- Usually, XML trees are modeled as labeled trees over a finite alphabet.
- For schema languages this is ok.
- For queries, the actual alphabet might depend on the query.
- Unary predicates on data values can be also modeled this way.

### Signatures
- There are various ways to represent unordered trees as finite relational models.
- Possible relations:
  - Local orders: $\rightarrow$, $\downarrow$
  - Their transitive closures: $\rightarrow^+$, $\downarrow^+$, $\leftarrow^+$, $\uparrow^+$
  - Reflexive and transitive: $\rightarrow^*$, $\downarrow^*$, $\leftarrow^*$, $\uparrow^*$
  - Document order: $\uparrow^* \rightarrow \downarrow^*$
- Frequent combinations:
  - $\rightarrow$, $\downarrow$
  - $\rightarrow^+$, $\downarrow^+$
  - $\rightarrow$, $\downarrow$, $\rightarrow^+$, $\downarrow^+$
  - $\rightarrow$, $\downarrow$, $\rightarrow^+$
## Contents

- Introduction
- **MSO Logics**
  - Automata and MSO-logic on Trees
  - Schema Languages
  - Node-selecting Queries
  - XML Transformations
- **Weaker Logics**
- Extensions
- Conclusion
Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits

Idea

Two states: $q_0, q_1$

$q_1 \equiv$ subtree evaluates to 1

Transitions

$\delta(\land, q_1) = \{(q_1, q_1)\}$
$\delta(\land, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$
$\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$
$\delta(\lor, q_0) = \{(q_0, q_0)\}$
$\delta(0, q_0) = \{\varepsilon\}; \delta(0, q_1) = \emptyset$
$\delta(1, q_1) = \{\varepsilon\}; \delta(1, q_0) = \emptyset$
Idea

Two states: $q_0, q_1$

$q_1 \equiv$ subtree evaluates to 1

Transitions

$\delta(\land, q_1) = \{(q_1, q_1)\}$

$\delta(\land, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

$\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\lor, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$
Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits

Idea

Two states: \( q_0, q_1 \)

\( q_1 \equiv \) subtree evaluates to 1

Transitions

\[ \delta(\wedge, q_1) = \{(q_1, q_1)\} \]
\[ \delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \]
\[ \delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \]
\[ \delta(\vee, q_0) = \{(q_0, q_0)\} \]
\[ \delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset \]
\[ \delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset \]
Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits

Idea

Two states: $q_0, q_1$

$q_1 \equiv$ subtree evaluates to 1

Transitions

$\delta(\land, q_1) = \{(q_1, q_1)\}$
$\delta(\land, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$
$\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$
$\delta(\lor, q_0) = \{(q_0, q_0)\}$
$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$
$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$
Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits

Idea

Two states: $q_0, q_1$
$q_1$ $\equiv$ subtree evaluates to 1

Transitions

$\delta(\land, q_1) = \{(q_1, q_1)\}$
$\delta(\land, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$
$\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$
$\delta(\lor, q_0) = \{(q_0, q_0)\}$
$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$
$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$
Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits

Idea
Two states: \( q_0, q_1 \)
\( q_1 \equiv \text{subtree evaluates to 1} \)

Transitions
\[
\begin{align*}
\delta(\wedge, q_1) &= \{(q_1, q_1)\} \\
\delta(\wedge, q_0) &= \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \\
\delta(\lor, q_1) &= \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \\
\delta(\lor, q_0) &= \{(q_0, q_0)\} \\
\delta(0, q_0) &= \{\epsilon\}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{\epsilon\}; \delta(1, q_0) = \emptyset 
\end{align*}
\]

Top-down Automaton for Boolean Circuits

Idea
Three states: \( q_0, q_1, \text{acc} \)
\( q_1 \equiv \text{subtree will be 1} \)

Transitions
\[
\begin{align*}
\delta(\wedge, q_1) &= \{(q_1, q_1)\} \\
\delta(\wedge, q_0) &= \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\} \\
\delta(\lor, q_1) &= \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\} \\
\delta(\lor, q_0) &= \{(q_0, q_0)\} \\
\delta(0, q_0) &= \{\text{acc}\}; \delta(0, q_1) = \emptyset \\
\delta(1, q_1) &= \{\text{acc}\}; \delta(1, q_0) = \emptyset 
\end{align*}
\]
Automata for Ranked Trees

**Bottom-up Automaton for (Tree-) Boolean Circuits**

Idea

Two states: $q_0, q_1$

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

**Top-down Automaton for Boolean Circuits**

Idea

Three states: $q_0, q_1, \text{acc}$

$q_1 \equiv$ subtree will be 1

Transitions

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

**Bottom-up Automaton for (Tree-) Boolean Circuits**

**Idea**

Two states: \(q_0, q_1\)

- \(q_1 \equiv \text{subtree evaluates to 1}\)

**Transitions**

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

**Top-down Automaton for Boolean Circuits**

**Idea**

Three states: \(q_0, q_1, \text{acc}\)

- \(q_1 \equiv \text{subtree will be 1}\)

**Transitions**

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

Bottom-up Automaton for (Tree-) Boolean Circuits

Idea
Two states: \( q_0, q_1 \)
\( q_1 \equiv \text{subtree evaluates to 1} \)

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

Top-down Automaton for Boolean Circuits

Idea
Three states: \( q_0, q_1, \text{acc} \)
\( q_1 \equiv \text{subtree will be 1} \)

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

Bottom-up Automaton for (Tree-) Boolean Circuits

Idea
Two states: $q_0, q_1$
$q_1 \equiv$ subtree evaluates to 1

Transitions

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

Top-down Automaton for Boolean Circuits

Idea
Three states: $q_0, q_1, \text{acc}$
$q_1 \equiv$ subtree will be 1

Transitions

$\delta(\land, q_1) = \{(q_1, q_1)\}$
$\delta(\land, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$
$\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$
$\delta(\lor, q_0) = \{(q_0, q_0)\}$
$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$
$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$
Automata for Ranked Trees

### Bottom-up Automaton for (Tree-) Boolean Circuits

#### Idea

Two states: $q_0, q_1$

$q_1 \equiv \text{subtree evaluates to 1}$

#### Transitions

- $\delta(\wedge, q_1) = \{(q_1, q_1)\}$
- $\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$
- $\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$
- $\delta(\lor, q_0) = \{(q_0, q_0)\}$
- $\delta(0, q_0) = \{\varepsilon\}; \delta(0, q_1) = \emptyset$
- $\delta(1, q_1) = \{\varepsilon\}; \delta(1, q_0) = \emptyset$

### Top-down Automaton for Boolean Circuits

#### Idea

Three states: $q_0, q_1, \text{acc}$

$q_1 \equiv \text{subtree will be 1}$

#### Transitions

- $\delta(\wedge, q_1) = \{(q_1, q_1)\}$
- $\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$
- $\delta(\lor, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$
- $\delta(\lor, q_0) = \{(q_0, q_0)\}$
- $\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$
- $\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$
### 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

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

(a) $\iff$ (c): Just the same thing, viewed in two different ways
Regular Tree Languages

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

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

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

Definition

Such an \( L \) is called **regular**
## Regular Tree Languages

### 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

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

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

### Observation

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

  \[
  \begin{array}{c}
  \forall, q_1 \\
  q_0 \quad q_1
  \end{array}
  \]

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

### Definition

Such an \( L \) is called **regular**
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

(a) $\Rightarrow$ (b): Powerset construction

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

Definition

Such an $L$ is called **regular**

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

```

     ∧
    / \   \
   /   \  /  \
  /     \ /  \
 q0 1 1 0 0 1 1 q1
```

M"unchen May 05 Logic and XML 17 Thomas Schwentick
**Definition: MSO logic**

- **MSO-formulas** talk about
  - node labels \( Q_0, Q_1, Q_\land, Q_\lor \)
  - Children and neighbors: \( \to, \downarrow \)
  - the root of the tree (root)

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

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

**Remark**

Exact signature does not matter

**Example: Boolean Circuits**

\[
\exists X\ X(\text{root}) \land \forall x
\]

\[
(Q_0(x) \to \neg X(x)) \land
\]

\[
((Q_\land(x) \land X(x)) \to (\forall y[(x \downarrow y) \to X(y)])) \land
\]

\[
((Q_\lor(x) \land X(x)) \to (\exists y[(x \downarrow y) \land X(y)]))
\]
### MSO and Regular Tree Languages

#### Definition: MSO logic

- **MSO-formulas** talk about
  - node labels \((Q_0, Q_1, Q_\land, Q_\lor)\)
  - Children and neighbors: \(\rightarrow, \downarrow\)
  - the root of the tree (root)

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

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

#### Remark

Exact signature does not matter

#### Example: Boolean Circuits

\[
\exists X \ X(\text{root}) \land \ \forall x \\
(Q_0(x) \rightarrow \neg X(x)) \land \\
((Q_\land(x) \land X(x)) \rightarrow (\forall y[(x \downarrow y) \rightarrow X(y)])) \land \\
((Q_\lor(x) \land X(x)) \rightarrow (\exists y[(x \downarrow y) \land X(y)])
\]

#### Theorem (Doner 70; Thatcher, Wright 68)

On ranked trees:

\[
\text{MSO} \equiv \text{Regular Tree Languages}
\]

#### Proof idea

**Automata \(\Rightarrow\) MSO:**

Formula expresses that there exists a correct tiling

**MSO \(\Rightarrow\) Automata:** more involved

Basic idea:

Automaton computes for each node \(v\)

the set of formulas which hold in the subtree rooted at \(v\)
MSO and Regular Tree Languages (cont.)

- 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 $t$
- $t_1 \equiv_k t_2 : k\text{-type}(t_1) = k\text{-type}(t_2)$
- Automaton computes $k$-type of tree and concludes whether $\varphi$ holds

Crucial fact:

$$
\begin{array}{c}
t_1 \equiv_k t_1' \\
t_2 \equiv_k t_2'
\end{array}
$$
On ranked trees, transitions are described by finite sets: 

$$\delta(\sigma, q) = \{(q_1,q_2),(q_3,q_4), \ldots\}$$
From Ranked to Unranked Trees

<table>
<thead>
<tr>
<th>Ranked trees</th>
<th>Unranked trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>On ranked trees, transitions are described by finite sets:</td>
<td></td>
</tr>
<tr>
<td>( \delta(\sigma, q) = {(q_1, q_2), (q_3, q_4), \ldots} )</td>
<td></td>
</tr>
</tbody>
</table>
| \[ \begin{array}{c}
q \\
\sigma_1 \\
q_1 \\
\sigma_2 \\
q_2 \\
\sigma \\
q \\
\end{array} \] | |
| \[ \begin{array}{c}
\sigma_1 \\
q_1 \\
\sigma_2 \\
q_2 \\
\vdots \\
\sigma_n \\
q_n \\
\end{array} \] | |

München May 05 Logic and XML 20 Thomas Schwentick
From Ranked to Unranked Trees

<table>
<thead>
<tr>
<th>Ranked trees</th>
<th>Unranked trees</th>
</tr>
</thead>
</table>
| On ranked trees, transitions are described by finite sets:  
\[ \delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\} \]  
\[ q \quad \sigma \]
|  \[ q_1 \quad \sigma_1 \]
|  \[ q_2 \quad \sigma_2 \]  
|  
| \[ q \quad \sigma \]
|  \[ q_1 \quad \sigma_1 \]
|  \[ q_2 \quad \sigma_2 \]
|  \[ \ldots \quad \ldots \]
|  \[ q_n \quad \sigma_n \]  
|  
|  \[ q_1 \quad q_2 \quad \ldots \quad q_n \]  
|  $\in \delta(\sigma, q)$?
From Ranked to Unranked Trees

**Ranked trees**

On ranked trees, transitions are described by finite sets:

\[ \delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots \} \]

**Unranked trees**

For unranked trees, \( \delta(\sigma, q) \) is a regular language

\( \delta(\sigma, q) \) can be specified by regular expression or finite string automaton

[Briegemann-Klein, Murata, Wood 2001]
### Theorem

The following are equivalent for a set $L$ of unranked trees:

1. **(a)** $L$ is accepted by a nondeterministic bottom-up automaton
2. **(b)** $L$ is accepted by a deterministic bottom-up automaton
3. **(c)** $L$ is accepted by a nondeterministic top-down automaton
4. **(d)** $L$ is characterized by an MSO-formula

### Definition

Again: such an $L$ is called **regular**

### Complexity issues for MSO on trees

#### Data Complexity:

- Query evaluation is possible in time $O(|t|)$

#### Combined Complexity:

- Query evaluation is complete for $\text{PSPACE}$
- Query evaluation is possible in time $f(|\varphi|)|t|$, where $f$ is $\sim 2^{2^{\cdot\cdot\cdot2^{2|\varphi|}}}$

- No elementary $f$ possible unless $\text{P} = \text{NP}$ [Frick, Grohe 2002]
- Satisfiability: $f(|\varphi|)$ (same $f$)
- In practice much better: MONA [Klarlund et al.]
Contents

Introduction

**MSO Logics**
- Automata and MSO-logic on Trees
- Schema Languages
- Node-selecting Queries
- XML Transformations

**Weaker Logics**

Extensions

Conclusion
DTDs and their Weakness

**DTDs**

- DTDs are essentially generalized context-free grammars

→ [Berstel, Boasson 00] provide characterizations

**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)>  
]
```
**DTDs and their Weakness**

### DTDs
- DTDs are essentially generalized context-free grammars
- 
  > [Berstel, Boasson 00] provide characterizations

### 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)> 
]> 
```

### Weakness of DTDs
- Elements with the same name may have different structure in different contexts
- It would be nice to have types for elements
- **Extended DTDs**

### A classical example

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

### Intention

```
Dealer
  UsedCars
    ad
      model
      year
  NewCars
    ad
      model
```
**Extended DTDs**

**Definition [Papakonstantinou, Vianu 2000]**

An extended DTD (EDTD) 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

**Example**

Dealer $\rightarrow$ UsedCars NewCars $\mu$(Dealer) $\equiv$ Dealer
UsedCars $\rightarrow$ adUsed* $\mu$(UsedCars) $\equiv$ UsedCars
NewCars $\rightarrow$ adNew* $\mu$(NewCars) $\equiv$ NewCars
adUsed $\rightarrow$ model year $\mu$(adUsed) $\equiv$ ad
adNew $\rightarrow$ model $\mu$(adNew) $\equiv$ ad

**Note**

Extended DTDs are often called *specialized DTDs*
Extended DTDs

Definition [Papakonstantinou, Vianu 2000]

An extended DTD (EDTD) 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

Example

Dealer $\rightarrow$ UsedCars NewCars $\mu$(Dealer) $=$ Dealer
UsedCars $\rightarrow$ adUsed* $\mu$(UsedCars) $=$ UsedCars
NewCars $\rightarrow$ adNew* $\mu$(NewCars) $=$ NewCars
adUsed $\rightarrow$ model year $\mu$(adUsed) $=$ ad
adNew $\rightarrow$ model $\mu$(adNew) $=$ ad

Note

Extended DTDs are often called *specialized DTDs*

EDTD for Boolean circuits

$$
\begin{align*}
1\text{-AND} & \rightarrow (1\text{-OR} \mid 1\text{-AND} \mid 1\text{-leaf})^* \\
1\text{-OR} & \rightarrow .* (1\text{-OR} \mid 1\text{-AND} \mid 1\text{-leaf}) .* \\
0\text{-AND} & \rightarrow .* (0\text{-OR} \mid 0\text{-AND} \mid 0\text{-leaf}) .* \\
0\text{-OR} & \rightarrow (0\text{-OR} \mid 0\text{-AND} \mid 0\text{-leaf})^* \\
1\text{-leaf} & \rightarrow \epsilon \\
0\text{-leaf} & \rightarrow \epsilon
\end{align*}
$$

<table>
<thead>
<tr>
<th>Tag</th>
<th>$\mu$(Tag)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-AND</td>
<td>AND</td>
</tr>
<tr>
<td>0-AND</td>
<td>AND</td>
</tr>
<tr>
<td>1-OR</td>
<td>OR</td>
</tr>
<tr>
<td>0-OR</td>
<td>OR</td>
</tr>
<tr>
<td>1-leaf</td>
<td>1</td>
</tr>
<tr>
<td>0-leaf</td>
<td>0</td>
</tr>
</tbody>
</table>
Extended DTDs and MSO

<table>
<thead>
<tr>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- A tree conforms to an extended DTD ((d, \mu)) if there is a labeling of its nodes by types which is valid wrt. (d).</td>
</tr>
</tbody>
</table>
## Extended DTDs and MSO

<table>
<thead>
<tr>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- A tree conforms to an extended DTD ((d, \mu)) if there is a labeling of its nodes by types which is valid wrt. (d)</td>
</tr>
<tr>
<td>- This reminds us of something...</td>
</tr>
</tbody>
</table>
## Observation

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

## Theorem

Extended DTDs capture exactly the regular tree languages
### Observation

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

### Theorem

Extended DTDs capture exactly the regular tree languages

### Remarks

- Regular tree languages and MSO-logic are a convenient framework for the study of XML schema languages
- Practical languages as XML Schema usually correspond to subclasses
- Full MSO power: Relax NG
- What about queries?
Contents

Introduction

MSO Logics

Automata and MSO-logic on Trees

Schema Languages

Node-selecting Queries

XML Transformations

Weaker Logics

Extensions

Conclusion
<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there an equally robust class of node-selecting queries?</td>
</tr>
</tbody>
</table>
## Node-selecting Queries

<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there an equally robust class of node-selecting queries?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a simple way to define node selecting queries by monadic second-order formulas:</td>
</tr>
<tr>
<td>Simply use one free variable: $\varphi(x)$</td>
</tr>
<tr>
<td>Is there a corresponding automaton model?</td>
</tr>
<tr>
<td>It is relatively easy to add node selection to nondeterministic bottom-up automata</td>
</tr>
</tbody>
</table>
Node-selecting Queries

Question

Is there an equally robust class of node-selecting queries?

Observations

● There is a simple way to define node selecting queries by monadic second-order formulas:
   ² Simply use one free variable: $\varphi(x)$
   ² Is there a corresponding automaton model?
   ² It is relatively easy to add node selection to nondeterministic bottom-up automata

Definition

● Nondeterministic 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 $t$ : there is an accepting computation on $t$ in which $v$ gets a state $q$ such that $s(q, \lambda(v)) = 1$
**Node-selecting Queries**

<table>
<thead>
<tr>
<th>Question</th>
<th>Existential vs. universal semantics</th>
</tr>
</thead>
</table>
| Is there an equally robust class of node-selecting queries? | - 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 |

<table>
<thead>
<tr>
<th>Observations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a simple way to define node selecting queries by monadic second-order formulas:</td>
<td></td>
</tr>
<tr>
<td>Simply use one free variable: $\varphi(x)$</td>
<td></td>
</tr>
<tr>
<td>Is there a corresponding automaton model?</td>
<td></td>
</tr>
<tr>
<td>It is relatively easy to add node selection to nondeterministic bottom-up automata</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition</th>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nondeterministic node-selecting automata</td>
<td>A node selecting query is MSO-definable iff it is expressible by a nondeterministic bottom-up node selecting automaton</td>
</tr>
<tr>
<td>Nondeterministic bottom-up automata plus select function: $s: Q \times \Sigma \rightarrow {0,1}$</td>
<td></td>
</tr>
<tr>
<td>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 $s(q,\lambda(v)) = 1$</td>
<td></td>
</tr>
</tbody>
</table>
Theorem

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
### Node-selecting queries (cont.)

#### More query models

- Same combined complexity as for language recognition (**PSPACE**-complete)

  → 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]

#### Data complexity

- **MSO node-selecting queries can be evaluated in two passes**

  **First pass, bottom-up:** Compute the types of the subtrees

  **Second pass, top-down:** Compute the types of the envelopes and combines it with the subtree information

  → 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: Data complexity is linear time
### Definition: XSLT TYPECHECKING

| Given: | DTDs $d_1$ and $d_2$  
Transformation $T$ |
|--------|------------------|
| Result: | Is $T(t) \models d_2$ for each document $t$  
with $t \models d_1$? |

### Questions
- Is XSLT TYPECHECKING decidable?
- What is the complexity?
### XML Transformations and MSO-Logic

**Definition: XSLT TypeChecking**

**Given:** DTDs $d_1$ and $d_2$
Transformation $T$

**Result:** Is $T(t) \models d_2$ for each document $t$ with $t \models d_1$?

**Questions**
- Is XSLT TypeChecking decidable?
- What is the complexity?

### Outline

- Provide an automata model for XSLT transformations
- Show that the behaviour of these automata can be captured by MSO logic
- Use manipulation of regular tree languages to solve type checking problem
- [Milo,Suciu,Vianu 01]
XSLT in more detail

How XSLT roughly works

Templates:  \(<\text{xsl:template name=TName match=pattern mode=MName}\\>\)

Template application:
  \(<\text{xsl:apply-templates select=Expression mode=MName}\\>\)

XSLT Processing  Whenever \text{xsl:apply-templates} is called at a node \(v\) the following happens:

- Compute set \(S(v)\) of nodes, reachable from \(v\) via \(Expression\) (if \(select\) is not present, \(S(v) =\) children of \(v\))
- For each \(w \in S(v)\) compute which templates can be applied to \(w\):
  - \(w\) has to match \text{pattern} of a template
  - \text{the mode} of the template has to be the same as the mode of \text{xsl:apply-templates}
- If no template matches, take the default template
- For each \(w \in S(v)\) select the best template and apply it.

The process starts at the root of the tree
## XSLT: Example

### Example Transformation

*Remove everything below a c. Translate a below b into d*

### Example XSLT

```xml
<xsl:template match="a">
  <a>
    <xsl:apply-templates/>
  </a>
</xsl:template>

<xsl:template match="a" mode="below">
  <d>
    <xsl:apply-templates/>
  </d>
</xsl:template>

<xsl:template match="b">
  <b>
    <xsl:apply-templates mode="below"/>
  </b>
</xsl:template>

<xsl:template match="b" mode="below">
  <b>
    <xsl:apply-templates mode="below"/>
  </b>
</xsl:template>

<xsl:template match="c">
  <c/>
</xsl:template>

<xsl:template match="c" mode="below">
  <c/>
</xsl:template>
```
XSLT: Example

Example Transformation

Remove everything below a \texttt{c}. Translate \texttt{a} below \texttt{b} into \texttt{d}

Example XSLT (Abbreviated)

\[
\begin{align*}
\langle... \text{ match}=&^\text{"a"}\rangle \langle a \rangle \langle xsl:apply-templates \rangle \langle /a \rangle \langle /...\rangle \\
\langle... \text{ match}=&^\text{"a"} \text{ mode}=&^\text{"below"}\rangle \langle d \rangle \langle xsl:apply-templates \rangle \langle /d \rangle \langle /...\rangle \\
\langle... \text{ match}=&^\text{"b"}\rangle \langle b \rangle \langle xsl:apply-templates \text{ mode}=&^\text{"below"}\rangle \langle /b \rangle \langle /...\rangle \\
\langle... \text{ match}=&^\text{"b"} \text{ mode}=&^\text{"below"}\rangle \langle b \rangle \langle xsl:apply-templates \text{ mode}=&^\text{"below"}\rangle \langle /b \rangle \langle /...\rangle \\
\langle... \text{ match}=&^\text{"c"}\rangle \langle c \rangle \langle /c \rangle \langle /...\rangle \\
\langle... \text{ match}=&^\text{"c"} \text{ mode}=&^\text{"below"}\rangle \langle c \rangle \langle /c \rangle \langle /...\rangle 
\end{align*}
\]
Example Transformation

Remove everything below a c. Translate a below b into d

Example XSLT (Abbreviated)

```xml
<... match="a"> a </xsl:apply-templates> </a> </...
<... match="a" mode="below"> d </xsl:apply-templates> </d> </...
<... match="b"> b </xsl:apply-templates mode="below"> b </xsl:apply-templates> </b> </...
<... match="b" mode="below"> b </xsl:apply-templates mode="below"> b </xsl:apply-templates> </b> </...
<... match="c"> c </xsl:apply-templates> </c> </...
<... match="c" mode="below"> c </xsl:apply-templates mode="below"> c </xsl:apply-templates> </c> </...
```

Example Trees

```
    a
   / \  
  a   a
 /   /  
a   b   a
 |   |   |
a   a   a  b
 |   |   |
a   a   a
```

M"unchen May 05 Logic and XML 33 Thomas Schwentick
Example Transformation

*Remove everything below a c. Translate a below b into d*

Example XSLT (Abbreviated)

```
<... match="a">a</xsl:apply-templates/>a</...
<... match="a" mode="below">d</xsl:apply-templates/>d</...
<... match="b">b</xsl:apply-templates mode="below"/>b</...
<... match="b" mode="below">b</xsl:apply-templates mode="below"/>b</...
<... match="c">c</c></...
<... match="c" mode="below">c</c></...
```

Example Trees

```
a
\(a\) \(a\) \(b\)  \(a\) \(c\) \\
\(a\) \(a\) \(b\)  \(a\) \(a\) \(b\) \\
\(a\) \(a\)
```

\(\Rightarrow\)

```
a
```
**Example Transformation**

*Remove everything below a c. Translate a below b into d*

**Example XSLT (Abbreviated)**

\[
\begin{align*}
\langle \ldots \text{match}=&"a" \rangle \langle a \rangle \langle \text{xsl:apply-templates} \rangle \langle /a \rangle \langle /\ldots \rangle \\
\langle \ldots \text{match}=&"a" \text{ mode}=&"below" \rangle \langle d \rangle \langle \text{xsl:apply-templates} \rangle \langle /d \rangle \langle /\ldots \rangle \\
\langle \ldots \text{match}=&"b" \rangle \langle b \rangle \langle \text{xsl:apply-templates \ mode}=&"below" \rangle \langle /b \rangle \langle /\ldots \rangle \\
\langle \ldots \text{match}=&"b" \text{ mode}=&"below" \rangle \langle b \rangle \langle \text{xsl:apply-templates \ mode}=&"below" \rangle \langle /b \rangle \langle /\ldots \rangle \\
\langle \ldots \text{match}=&"c" \rangle \langle c \rangle \langle /c \rangle \langle /\ldots \rangle \\
\langle \ldots \text{match}=&"c" \text{ mode}=&"below" \rangle \langle c \rangle \langle /c \rangle \langle /\ldots \rangle
\end{align*}
\]

**Example Trees**

```
\( a \quad \)  \\
| \quad |  \\
\( a \quad a \quad b \quad a \quad a \quad b \quad c \quad a \quad b \quad a \quad c \quad \Rightarrow \quad a \quad b \quad a \quad c \quad \)
```
XSLT: Example

Example Transformation

Remove everything below a \textit{c}. Translate \textit{a} below \textit{b} into \textit{d}

Example XSLT (Abbreviated)

\[
\begin{align*}
\langle \text{... match}=\"a\" \rangle \langle a \rangle \langle xsl:apply-templates \rangle \langle /a \rangle \langle /... \rangle \\
\langle \text{... match}=\"a\" \text{ mode}=\"below\" \rangle \langle d \rangle \langle xsl:apply-templates \rangle \langle /d \rangle \langle /... \rangle \\
\langle \text{... match}=\"b\" \rangle \langle b \rangle \langle xsl:apply-templates \text{ mode}=\"below\" \rangle \langle /b \rangle \langle /... \rangle \\
\langle \text{... match}=\"b\" \text{ mode}=\"below\" \rangle \langle b \rangle \langle xsl:apply-templates \text{ mode}=\"below\" \rangle \langle /b \rangle \langle /... \rangle \\
\langle \text{... match}=\"c\" \rangle \langle c \rangle \langle /c \rangle \langle /... \rangle \\
\langle \text{... match}=\"c\" \text{ mode}=\"below\" \rangle \langle c \rangle \langle /c \rangle \langle /... \rangle
\end{align*}
\]

Example Trees

\[
\begin{align*}
\begin{array}{c}
a \\
a \quad b \\
\quad a \quad a \quad b \\
\quad a \quad a \quad a \quad b \\
\quad a \quad a \quad a \quad b \quad c \\
\end{array} & \Rightarrow \\
\begin{array}{c}
a \\
a \quad b \\
a \quad a \quad c \\
a \quad a \quad d \quad b \quad a \\
\end{array}
\end{align*}
\]
Example Transformation

Remove everything below a c. Translate a below b into d

Example XSLT (Abbreviated)

```xml
<... match="a"> a <xsl:apply-templates> /a </a> </...
<... match="a" mode="below"> d <xsl:apply-templates> /d </d> </...
<... match="b"> b <xsl:apply-templates mode="below"> /b </b> </...
<... match="b" mode="below"> b <xsl:apply-templates mode="below"> /b </b> </...
<... match="c"> c </c> </...
<... match="c" mode="below"> c </c> </...
```

Example Trees

```
```

```
```
### Definition: $k$-pebble Transducer

- Work on binary tree encodings of unranked trees
- Up to $k$ pebbles can be placed on the tree
- 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 child, right child or parent
  - lift current pebble
  - place new pebble on the root
- Nondeterminism allowed
  
  (Proof presented here: deterministic case)

- If current pebble stays it is possible to produce output:
  - a node with two (forthcoming) subtrees; in this case two independent subcomputations (branches) are started, which construct the left subtree and right subtree, respectively
  - a leaf; in this case the computation branch stops
Back to the Typechecking Question

Proof idea

• How can we check that $T(t) \in L(d_2)$, for each $t \in L(d_1)$?

• 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:

\[
\begin{array}{c}
  b \\
  \downarrow \\
  a & a & a & a \\
\end{array}
\Rightarrow
\begin{array}{c}
  b \\
  \downarrow \\
  a & a \\
  \downarrow \\
  a & a \\
\end{array}
\]

• Better approach:
  – Compute $T^{-1}(L(d_2))$
  – Check $L(d_1) \cap T^{-1}(L(d_2)) = \emptyset$
Back to the Typechecking Question

Proof idea

- How can we check that $T(t) \in L(d_2)$, for each $t \in L(d_1)$?
- 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:

Better approach:

- Compute $T^{-1}(L(d_2))$
- Check $L(d_1) \cap T^{-1}(L(d_2)) = \emptyset$

Definition: $k$-pebble acceptors

- Basically the same as $k$-pebble transducers
- Instead of output producing steps:
  - accept
  - branch into two independent subcomputations
- A tree is accepted if all subcomputations accept

Main Steps of the Proof

(i) $T$ computed by $k$-pebble transducer

$$\Rightarrow T = T_1 \circ \cdots \circ T_{k+1}$$

with 0-pebble transducers $T_i$

[Engelfriet, Maneth 03]

(ii) $L$ regular, $T_i$ 0-pebble transducer

$$\Rightarrow T_i^{-1}(L)$$

accepted by 0-pebble acceptor

(iii) 0-pebble acceptors only accept regular tree languages
Step (ii)

Lemma

\( L \) regular, \( T_i \) 0-pebble transducer

\[ \Rightarrow T_i^{-1}(L) \] accepted by 0-pebble acceptor

Proof

- Let \( B \) be a nondeterministic top-down tree automaton which accepts \( \overline{L} \)
- We construct a 0-pebble acceptor \( A \) for \( T_i^{-1}(\overline{L}) \), i.e., an automaton which on input \( t \) decides whether \( T(t) \) is accepted by \( B \):
  - Simulate \( T \) on \( t \) and \( B \)
  - Simulate at the same time the behaviour of \( B \) on the (virtual) output tree - this is possible as the output tree is produced top-down and can be instantly consumed by \( B \)
  - The simulation involves branching, whenever \( T \) branches
<table>
<thead>
<tr>
<th>Lemma</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pebble acceptors only accept regular tree languages</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proof idea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:</td>
</tr>
<tr>
<td>1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)</td>
</tr>
<tr>
<td>2. Show that AGAP can be expressed in MSO</td>
</tr>
</tbody>
</table>
**Step (iii)**

**Lemma**

0-pebble acceptors only accept regular tree languages

**Proof idea**

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

**Accessible Nodes**

Let \( G = (V,E), V = V_\wedge \cup V_\lor \). A node \( w \) is **accessible** if

- \( w \in V_\wedge \) and all successors of \( w \) are accessible, or
- \( w \in V_\lor \) and at least one successor of \( w \) is accessible

**Definition: AGAP**

**Given:**

Graph \( G = (V,E) \), \( V = V_\wedge \cup V_\lor \), and \( v \in V \)

**Question:**

Is \( v \) accessible?

**Example**

![Example diagram showing AGAP](image)
Step (iii)

Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Definition: AGAP

Given: Graph $G = (V,E)$, $V = V_\land \cup V_\lor$, and $v \in V$

Question: Is $v$ accessible?

Example

Accessible Nodes

Let $G = (V,E)$, $V = V_\land \cup V_\lor$. A node $w$ is accessible if

- $w \in V_\land$ and all successors of $w$ are accessible, or
- $w \in V_\lor$ and at least one successor of $w$ is accessible
**Step (iii)**

<table>
<thead>
<tr>
<th>Lemma</th>
<th>Definition: AGAP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong>-pebble acceptors only accept regular tree languages</td>
<td><strong>Given:</strong> Graph $G = (V,E)$, $V = V_\wedge \cup V_\vee$, and $v \in V$</td>
</tr>
<tr>
<td><strong>Proof idea</strong></td>
<td><strong>Question:</strong> Is $v$ accessible?</td>
</tr>
<tr>
<td>Show that the language of <strong>0</strong>-pebble acceptors can be expressed by an MSO-formula:</td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>1. Reduce <strong>0</strong>-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)</td>
<td>![Diagram]</td>
</tr>
<tr>
<td>2. Show that AGAP can be expressed in MSO</td>
<td></td>
</tr>
</tbody>
</table>

**Accessible Nodes**

Let $G = (V,E)$, $V = V_\wedge \cup V_\vee$. A node $w$ is **accessible** if

- $w \in V_\wedge$ and all successors of $w$ are accessible, or
- $w \in V_\vee$ and at least one successor of $w$ is accessible
Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Definition: AGAP

Given:
Graph \( G = (V, E) \), \( V = V^\wedge \cup V^\lor \), and \( v \in V \)

Question: Is \( v \) accessible?

Accessible Nodes

Let \( G = (V, E) \), \( V = V^\wedge \cup V^\lor \). A node \( w \) is accessible if

- \( w \in V^\wedge \) and all successors of \( w \) are accessible, or
- \( w \in V^\lor \) and at least one successor of \( w \) is accessible
Step (iii)

Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Accessible Nodes

Let $G = (V,E)$, $V = V_\land \cup V_\lor$. A node $w$ is accessible if

- $w \in V_\land$ and all successors of $w$ are accessible, or
- $w \in V_\lor$ and at least one successor of $w$ is accessible

Definition: AGAP

Given: Graph $G = (V,E)$, $V = V_\land \cup V_\lor$, and $v \in V$

Question: Is $v$ accessible?

Example
### Lemma

**0**-pebble acceptors only accept regular tree languages

### Proof idea

Show that the language of **0**-pebble acceptors can be expressed by an MSO-formula:

1. Reduce **0**-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

### Definition: AGAP

**Given:** Graph \( G = (V,E) \), \( V = V_\wedge \cup V_\vee \), and \( v \in V \)

**Question:** Is \( v \) accessible?

### Accessible Nodes

Let \( G = (V,E) \), \( V = V_\wedge \cup V_\vee \). A node \( w \) is accessible if

- \( w \in V_\wedge \) and all successors of \( w \) are accessible, or
- \( w \in V_\vee \) and at least one successor of \( w \) is accessible

### Example

![Example Diagram]
Construction of $G_{A,t}$

- Nodes in $V_\vee$ are the configurations of $A$ on $t$: pairs $[q,v]$, state $q$, node $v$ of $t$
- Nodes in $V_\wedge$ are $\epsilon$ and certain pairs $(\gamma_1,\gamma_2)$ of configurations
- Edges:
  - $(\gamma_1,\gamma_2) \rightarrow \gamma_1$, $(\gamma_1,\gamma_2) \rightarrow \gamma_2$
  - $\gamma \rightarrow \gamma'$, if this is a step of $A$
  - $\gamma \rightarrow \epsilon$, if $A$ can get into the accept state from $\gamma$
  - $\gamma \rightarrow (\gamma_1,\gamma_2)$ if this is a branching step of $A$

Facts

- $A$ accepts $t \iff [q,\text{root}]$ is accessible in $G_{A,t}$ with $q \in F$
- $|G_{A,t}| = O(|t|)$
### Construction of $G_{A,t}$

- Nodes in $V_\vee$ are the configurations of $A$ on $t$:
  - pairs $[q,v]$, state $q$, node $v$ of $t$
- Nodes in $V_\wedge$ are $\epsilon$ and certain pairs $(\gamma_1,\gamma_2)$ of configurations
- Edges:
  - $(\gamma_1,\gamma_2) \rightarrow \gamma_1$, $(\gamma_1,\gamma_2) \rightarrow \gamma_2$
  - $\gamma \rightarrow \gamma'$, if this is a step of $A$
  - $\gamma \rightarrow \epsilon$, if $A$ can get into the accept state from $\gamma$
  - $\gamma \rightarrow (\gamma_1,\gamma_2)$ if this is a branching step of $A$

### Facts

- $A$ accepts $t$ if $[q,\text{root}]$ is accessible in $G_{A,t}$ with $q \in F$
- $|G_{A,t}| = O(|t|)$

### Definition: Reverse-closed Sets of Nodes

A set $S$ of nodes is reverse-closed if:

- if $v$ is in $V_\wedge$ and $w \in S$, for all nodes $w$ with $(v,w) \in E$, then $v \in S$
- if $v$ is in $V_\vee$ and $w \in S$, for some node $w$ with $(v,w) \in E$, then $v \in S$

### Example

Node $v$ is accessible iff it is in every reverse-closed set of nodes

### Reverse-closed as MSO-Formula

$$\forall S \ (\text{rc}(S) \rightarrow S(v)), \text{ where } \text{rc}(S) \text{ is}$$

$$\forall x (\left[ V_\wedge(x) \land \forall y (E(x,y) \rightarrow S(y)) \right] \rightarrow S(x)) \land$$

$$\left[ V_\vee(x) \land \exists y (E(x,y) \land S(y)) \right] \rightarrow S(x))$$
Summary of proof

- Given \( d_1, d_2 \) and \( T \), we can proceed as follows:
  1. Construct the \( k \)-pebble acceptor \( A \) for \( T^{-1}(L(d_2)) \)
  2. Transform \( A \) into an equivalent MSO formula \( \Phi \)
  3. \( \Phi \) holds for all trees \( t \) for which \( T(t) \not\subseteq L(d_2) \)
  4. Construct a nondeterministic bottom-up automaton \( A' \) equivalent to \( \neg \Phi \)
  5. Check that \( L(d_1) \subseteq L(A') \)

- Complexity: non-elementary
### Summary of proof

- **Given** $d_1$, $d_2$ and $T$, we can proceed as follows:
  1. Construct the $k$-pebble acceptor $A$ for $T^{-1}(L(d_2))$
  2. Transform $A$ into an equivalent MSO formula $\Phi$
  3. $\Phi$ holds for all trees $t$ for which $T(t) \not\in L(d_2)$
  4. Construct a nondeterministic bottom-up automaton $A'$ equivalent to $\neg\Phi$
  5. Check that $L(d_1) \subseteq L(A')$

- **Complexity**: non-elementary

### Related work

- **TYPECHECKING is decidable for compositions of macro tree transducers** [Engelfriet, Maneth 03]

- If transformations are allowed to compare data values in the input document, type checking becomes undecidable very quickly, even for restricted types and transformations [Alon et al. 01]

- Typechecking for deterministic top-down tree transducers is more tractable. Complexity depends on exact representation of DTDs and restrictions on the transducers: between **PTIME** and **EXPTIME** [Martens, Neven 03]
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction</strong></td>
</tr>
<tr>
<td><strong>MSO Logics</strong></td>
</tr>
<tr>
<td><strong>Weaker Logics</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Temporal Logics</td>
</tr>
<tr>
<td>First-Order Logics</td>
</tr>
<tr>
<td>Transitive-Closure Logics</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
</tr>
<tr>
<td><strong>Conclusion</strong></td>
</tr>
<tr>
<td>Remarks</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>• MSO-logic offers a framework for dealing with schemas</td>
</tr>
<tr>
<td>• Close connection to automata</td>
</tr>
<tr>
<td>• Helpful in proofs</td>
</tr>
<tr>
<td>• Extends to (node-selecting) queries</td>
</tr>
<tr>
<td>• But: for many querying tasks MSO-power is not needed</td>
</tr>
<tr>
<td>• <strong>XPath</strong>: Navigation along paths</td>
</tr>
<tr>
<td>→ Similarity to temporal logics</td>
</tr>
<tr>
<td>• Presentation follows [Libkin 05]</td>
</tr>
</tbody>
</table>
Definition: An LTL-like logic: $\text{TL}^{\text{tree}}$

- $A, \varphi \lor \psi, \neg \varphi$
- $X_* \varphi, X_{\neg} \varphi, \varphi U_* \psi, \varphi U_{\neg} \psi$

($* = \rightarrow$ or $\downarrow$)

Theorem [Marx 04]

A unary or binary query over unordered trees is FO-definable (with $\downarrow^*, \rightarrow^*$) iff it is definable in $\text{TL}^{\text{tree}}$.

Proof idea

Proof similar as equivalence of LTL and FO on orders [Kamp 68]
### Navigation and Temporal Logics (cont.)

**Definition: An LTL-like logic: TL\(^{\text{tree}}\)**
- \(A, \varphi \lor \psi, \neg \varphi\)
- \(X^* \varphi, X^* \neg \varphi, \varphi U^* \psi, \varphi U^* \neg \psi\)

\((* = \rightarrow \text{ or } \downarrow)\)

**Theorem [Marx 04]**
A unary or binary query over unordered trees is FO-definable (with \(\downarrow^*\), \(\rightarrow^*\)) iff it is definable in TL\(^{\text{tree}}\)

**Proof idea**

Proof similar as equivalence of LTL and FO on orders [Kamp 68]

---

**Definition: A CTL\(^*\)-like logic: CTL\(^*\)\(_{\text{past}}\)**
- **Node formulas:**
  - \(A, \alpha \lor \alpha', \neg \alpha\)
  - \(E \beta^*\)
- **Path formulas:**
  - \(\alpha, \neg \beta^*, \beta \lor \beta'\)
  - \(X^* \beta, X^* \neg \beta, \beta U^* \beta', \beta U^* \neg \beta'\)

\((* = \rightarrow \text{ or } \downarrow)\)

**Theorem [Barcelo, Libkin 05]**
A unary or binary query over unordered trees is FO-definable (with \(\downarrow^*\), \(\rightarrow^*\)) iff it is definable in CTL\(^*\)\(_{\text{past}}\)

**Proof idea**

Proof similar as equivalence of CTL\(^*\) and FO on binary trees [Hafer, Thomas 87]
### Theorem [Marx 04]

Containment of Navigational \textsc{XPath} queries in the presence of DTDs is in \textit{EXPTIME}

### Proof idea

- Navigational \textsc{XPath} can be translated into propositional dynamic logic (PDL) (over structures with $\downarrow$, $\rightarrow$)
- DTDs can also be expressed by PDL-formulas
- Implication for PDL is \textit{EXPTIME}-complete

### Remark

For much weaker fragments, containment is already \textit{EXPTIME}-hard
Contents

Introduction

MSO Logics

**Weaker Logics**

Temporal Logics

First-Order Logics

Transitive-Closure Logics

Extensions

Conclusion
### XPath and Fragments of First-order Logic

#### Characterizations of XPath
- Navigational XPath (without `not` and `and`) corresponds to positive existential first-order logic.
- Different XPath axes correspond to different signatures.
  
  \[\text{[Benedikt, Fan, Kuper 03]}\]

#### Proof idea
- Basic idea: For each node $u$ of the query tree: guess a node $h(u)$ in the document tree and check that $h$ is a “homomorphism”.
- Main difficulty in proof: Deal with conjunctions of conditions.

#### Further Results on
- closure properties
- axiomatizations of equivalence
### XPath and Fragments of First-order Logic

#### Characterizations of XPath
- Navigational XPath (without not and and) corresponds to positive existential first-order logic
- Different XPath axes correspond to different signatures
  - [Benedikt, Fan, Kuper 03]

#### Proof idea
- Basic idea:
  - For each node $u$ of the query tree: guess a node $h(u)$ in the document tree and check that $h$ is a “homomorphism”
- Main difficulty in proof:
  - Deal with conjunctions of conditions

#### Further Results on
- closure properties
- axiomatizations of equivalence

---

<table>
<thead>
<tr>
<th>Theorem [Marx, de Rijke 05]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node-selecting navigational XPath queries correspond to unary two-variable first-order formulas over $\downarrow, \downarrow^<em>, \rightarrow, \rightarrow^</em>$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proof idea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proof follows the same lines as characterization of unary temporal logic by two-variable logic</td>
</tr>
</tbody>
</table>

---
<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is needed to capture full first-order logic?</td>
</tr>
</tbody>
</table>
### XPath and Full First-Order Logic

**Question**

What is needed to capture full first-order logic?

<table>
<thead>
<tr>
<th>Conditional XPath</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conditional axes</strong>: Expressions of the kind $P^+$, where $P$ is a location step</td>
</tr>
<tr>
<td><strong>Conditional XPath</strong>: Navigational XPath plus conditional axes</td>
</tr>
</tbody>
</table>

**Example**

$$(\text{child :: } a[\text{desc :: } b \text{ or child :: } c])^+$$

holds between $u$ and $v$ if

- $v$ is a descendant of $u$ and
- all intermediate nodes
  - are labelled with $a$ and
  - have a $c$-child or a $b$-descendant
What is needed to capture full first-order logic?

**Conditional XPath**

- **Conditional axes**: Expressions of the kind $P^+$, where $P$ is a location step.
- **Conditional XPath**: Navigational XPath plus conditional axes.

**Example**

$$(\text{child :: } a[\text{desc :: } b \text{ or child :: } c])^+$$

holds between $u$ and $v$ if

- $v$ is a descendant of $u$ and
- all intermediate nodes
  - are labelled with $a$ and
  - have a $c$-child or a $b$-descendant.

**Theorem [Marx 04,05]**

Conditional XPath corresponds exactly to first-order logic over $\downarrow^*$, $\rightarrow^*$ (wrt node-selecting and binary queries).
First-Order Logic Plus Regular Expressions

First-Order Logic

- vertical regular expressions (over paths)

- horizontal regular expressions (over children)

- Nesting of formulas and regular expressions
  \[
  [\varphi_1(s,t) \cdot \varphi_2(s,t)^* \cdot \varphi_3(s,t)]_{s,t}(x,y)
  \]
First-Order Logic and Automata

- MSO-logic ≡ Parallel tree automata
- FO-logic ≡ ???
Definition: Tree-walk automaton

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.

Illustration
First-Order Logic and Automata

- MSO-logic \equiv Parallel tree automata
- FO-logic \equiv ???

**Definition: Tree-walk automaton**

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

**Illustration**

![Tree-walk automaton diagram]

**Fact [Neven, Sch. 00]**

FO-sentences over binary trees with \(\downarrow\) and \(\rightarrow\)
can be evaluated by deterministic Tree-Walk automata

**Proof idea**

- Simple application of Gaifman’s Theorem
- Does not hold for unranked trees
First-Order Logic and Automata

- MSO-logic $\equiv$ Parallel tree automata
- FO-logic $\equiv$ ???

**Definition: Tree-walk automaton**

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

**Illustration**

![Tree-walk automaton illustration]

**Fact [Neven, Sch. 00]**

FO-sentences over binary trees with $\downarrow$ and $\rightarrow$ can be evaluated by deterministic Tree-Walk automata

**Proof idea**

- Simple application of Gaifman’s Theorem
- Does not hold for unranked trees

**Remark**

To capture the expressive power of Tree-walk automata one needs a bit more...

**Theorem [Neven, Sch. 00]**

- Nondet. TWA $\equiv TC^1[FO[\downarrow, \text{mod}]]$
- Det. TWA $\equiv DTC^1[FO[\downarrow, \text{mod}]]$

over binary trees with $\downarrow$ and $\rightarrow$
Contents

Introduction

MSO Logics

**Weaker Logics**

Temporal Logics

First-Order Logics

Transitive-Closure Logics

Extensions

Conclusion
## Remarks

- On strings: $MSO \equiv FO + \text{ unary TC}$
- On (binary) trees???
## Remarks

- On strings: $\text{MSO} \equiv \text{FO} + \text{unary TC}$
- On (binary) trees???

## Theorem [Engelfriet, Hoogeboom 05]

Nondeterministic pebble automata correspond to FO plus positive unary TC on binary trees

## Corollary

Positive, unary TC-logic is weaker than MSO on binary trees iff pebble automata are weaker than parallel automata
## MSO-logic vs. Unary Transitive Closure Logic

<table>
<thead>
<tr>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>• On strings: MSO $\equiv$ FO $+$ unary TC</td>
</tr>
</tbody>
</table>
| • On (binary) trees???

### Theorem [Engelfriet, Hoogeboom 05]

Nondeterministic pebble automata correspond to FO plus positive unary TC on binary trees

### Corollary

Positive, unary TC-logic is weaker than MSO on binary trees iff pebble automata are weaker than parallel automata

### Inclusion structure

<table>
<thead>
<tr>
<th>MSO = Pebble-Marble-TWA = ATWA</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{FO(posTC}^1) = \text{Pebble-TWA} ]</td>
</tr>
<tr>
<td>[ \text{FOREG} \neq ]</td>
</tr>
<tr>
<td>[ \text{NTWA} = \text{TC}^1[\text{FO}[\downarrow,\text{mod}]] ]</td>
</tr>
<tr>
<td>[ \text{DTWA} = \text{DTC}^1[\text{FO}[\downarrow,\text{mod}]] ]</td>
</tr>
<tr>
<td>[ \text{FO}^{\downarrow}[\downarrow^*] \neq ]</td>
</tr>
<tr>
<td>[ \text{FO}^{\downarrow} \neq ]</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
</tr>
<tr>
<td>MSO Logics</td>
</tr>
<tr>
<td>Weaker Logics</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
</tr>
<tr>
<td>Counting</td>
</tr>
<tr>
<td>Data Values</td>
</tr>
<tr>
<td>Conclusion</td>
</tr>
</tbody>
</table>
Extensions

So far...

- We have seen logics for
  - Validation, Typing
  - Navigation
  - Transformation
- What about more general queries?
  - results of higher arity?
  - joins, i.e., comparisons of data values
  - counting
## Extensions

<table>
<thead>
<tr>
<th>So far...</th>
<th>Counting</th>
</tr>
</thead>
</table>
| - We have seen logics for  
  - Validation, Typing  
  - Navigation  
  - Transformation  
- What about more general queries?  
  - results of higher arity?  
  - joins, i.e., comparisons 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:  
  - \( \equiv \) EMSO logic  
  - Whether automaton accepts all trees is 
    undecidable  
- Det. Presburger automata:  
  - \( \equiv \) Presburger \( \mu \)-formulas  
  - Membership test: \( O(|A||t|) \)  
  - Non-emptiness: \( \text{PSPACE} \)  
  - Containment: \( \text{PSPACE} \) |

[Seidl, Sch., Muscholl, Habermehl 2004]
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction</strong></td>
</tr>
<tr>
<td><strong>MSO Logics</strong></td>
</tr>
<tr>
<td><strong>Weaker Logics</strong></td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
</tr>
<tr>
<td>Counting</td>
</tr>
<tr>
<td>Data Values</td>
</tr>
<tr>
<td><strong>Conclusion</strong></td>
</tr>
</tbody>
</table>
## Data Values

<table>
<thead>
<tr>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>• In (database) queries or constraints comparisons of data values are very common</td>
</tr>
<tr>
<td>• Most XML theory concentrates on structure of trees instead</td>
</tr>
</tbody>
</table>

### Example queries/constraints

<table>
<thead>
<tr>
<th>• Did we reserve a room for every participant?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Does every participant give at most one talk?</td>
</tr>
<tr>
<td>•</td>
</tr>
</tbody>
</table>
## Data Values

### Remark
- In (database) queries or constraints comparisons of data values are very common.
- Most XML theory concentrates on structure of trees instead.

### Example queries/constraints
- Did we reserve a room for every participant?
- $\forall x \text{ Partic.Name}(x) \rightarrow \exists y \text{ Room.Name}(y) \land x.data = y.data$
- Does every participant give at most one talk?

Data Values

Remark

- In (database) queries or constraints comparisons of data values are very common
- Most XML theory concentrates on structure of trees instead

Example queries/constraints

- Did we reserve a room for every participant?
  
  \( \forall x \ Partic.Name(x) \rightarrow \exists y \ Room.Name(y) \land x.data = y.data \)

- Does every participant give at most one talk?
  
  \( \forall x \forall y \ [x \neq y \land \text{Talk.Speaker}(x) \land \text{Talk.Speaker}(y)] \rightarrow x.data \neq y.data \)
## Data Values

<table>
<thead>
<tr>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>• In (database) queries or constraints comparisons of data values are very common</td>
</tr>
<tr>
<td>• Most XML theory concentrates on structure of trees instead</td>
</tr>
</tbody>
</table>

### Example queries/constraints

- **Did we reserve a room for every participant?**
  \[
  \forall x \text{ Partic.Name}(x) \rightarrow \exists y \text{ Room.Name}(y) \land x.\text{data} = y.\text{data}
  \]

- **Does every participant give at most one talk?**
  \[
  \forall x \forall y [x \neq y \land \text{Talk.Speaker}(x) \land \text{Talk.Speaker}(y)] \rightarrow x.\text{data} \neq y.\text{data}
  \]

### The setting

- **We concentrate on data strings**
- **Access to data values only via equality tests**

### Example: data string

\[
\begin{align*}
2 & \ 3 & \ 3 & \ 3 & \ 2 & \ 2 & \ 7 & \ 17 & \ 17 & \ 3 & \ 4 & \ 5 & \ 2 & \ 3 & \ 3 & \ 4 & \ 4 \\
& c & b & c & a & a & b & b & b & c & a & b & a & c & b & a & a
\end{align*}
\]

### Definition

- **Data string**: Finite sequence over \( \Sigma \times \mathcal{D} \), where
  - \( \Sigma \) finite
  - \( \mathcal{D} \) infinite

- **Logical language**:
  - \( a(x) \): Letter position \( x \) is \( a \in \Sigma \)
  - order relation \( < \), successor relation \( +1 \)
  - \( \sim \): \( x \sim y \) if positions \( x \) and \( y \) have the same \( \mathcal{D} \)-value
  - \( \rightarrow \): Equivalence relation
Some Known Results about $\text{FO}^2$

<table>
<thead>
<tr>
<th>Over arbitrary relational structures</th>
<th>Over strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Finite model property [Mortimer 75]</td>
<td>• Satisfiability $\text{NEXPTIME}$-complete</td>
</tr>
<tr>
<td>• Satisfiability $\text{NEXPTIME}$-complete [Grädel et al. 97]</td>
<td>• Expressive power:</td>
</tr>
<tr>
<td>• On structures with 1 or 2 equivalence relations: decidable [Kieroński, Otto 05]</td>
<td>• unary LTL and $\Sigma^2 \cap \Pi^2$ [Etessami, Vardi, Wilke 97]</td>
</tr>
<tr>
<td>• On structures with 3 equivalence relations: undecidable [Kieroński, Otto 05]</td>
<td>• variety DA [Thérien, Wilke 98]</td>
</tr>
<tr>
<td>• On structures with a linear order: in $\text{coNEXPTIME}$ [Otto]</td>
<td>• two way, partially-ordered DFA [Sch., Thérien, Vollmer 01]</td>
</tr>
<tr>
<td>• On structures with several well-orderings: undecidable [Otto]</td>
<td></td>
</tr>
</tbody>
</table>
### Models

- **Register-Automata**: finitely many registers for data values
- **Pebble-Automata**: finitely many pebbles with stack discipline
- **Variations**: 1-way or 2-way, Deterministic or Nondeterministic

### Example

```
*  #1 ← σ  *  σ = #1
a → b → c
```

### Facts

- Non-emptiness undecidable for most models
- Exception: 1N-register automata
- This holds also for FO logic

→ [Kaminski, Francez 94; Neven, Sch., Vianu 01]
Overview of Automata Models

Inclusion Structure

- **1D-RA**
- **1N-RA**
- **2D-RA**
- **2N-RA**
- **2A-RA**
- **FO**
- **MSO**
- **W1D-PA**
- **W1N-PA**
- **2D-PA**
- **2N-PA**
- **S1D-PA**
- **S1N-PA**
- **2A-PA**

Relationships:
- 1D-RA ⊈ 1N-RA
- 1N-RA ⊈ 2N-RA
- 2N-RA ⊈ 2A-RA
- MSO ⊈ FO
- W1D-PA ⊈ W1N-PA
- 2D-PA ⊈ 2N-PA
- S1D-PA ⊈ S1N-PA
- 2A-PA ⊈ FO

The relationships are represented by dotted lines with symbols indicating inclusion or non-inclusion.
## Decidability Results

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Satisfiability of $\text{FO}^2(+1, &lt;)$ on data strings is decidable</th>
</tr>
</thead>
</table>

### Further results
- Emptiness of multicounter automata can be reduced to satisfiability of $\text{FO}^2(+1, <)$ on data strings
- $\text{FO}^2(<)$ on data strings is complete for $\text{NEXPTIME}$
- $\text{FO}^2(+1)$ on data strings is hard for $\text{NEXPTIME}$ (and in $2\text{NEXPTIME}$)
- $\text{FO}^3(+1)$ on data strings is undecidable
Decidability Results

Theorem

Satisfiability of $\text{FO}^2(+1, <)$ on data strings is decidable

Further results

- Emptiness of multicounter automata can be reduced to satisfiability of $\text{FO}^2(+1, <)$ on data strings
- $\text{FO}^2(<)$ on data strings is complete for $\text{NEXPTIME}$
- $\text{FO}^2(+1)$ on data strings is hard for $\text{NEXPTIME}$ (and in $2\text{NEXPTIME}$)
- $\text{FO}^3(+1)$ on data strings is undecidable

Some Definitions

- **Data string $s$:**
- **Class**: all positions with the same data value
- **Interval**: (maximal set of) contiguous positions of a class
- **String projection $P(s)$**: 

![Diagram](image_url)
Decidability Results

<table>
<thead>
<tr>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satisfiability of $\text{FO}^2(+1, &lt;)$ on data strings is decidable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Further results</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Emptiness of multicounter automata can be reduced to satisfiability of $\text{FO}^2(+1, &lt;)$ on data strings</td>
</tr>
<tr>
<td>• $\text{FO}^2(\leq)$ on data strings is complete for $\text{NEXPTIME}$</td>
</tr>
<tr>
<td>• $\text{FO}^2(+1)$ on data strings is hard for $\text{NEXPTIME}$ (and in $2\text{NEXPTIME}$)</td>
</tr>
<tr>
<td>• $\text{FO}^3(+1)$ on data strings is undecidable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Some Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Data string $s$:</td>
</tr>
<tr>
<td>• <strong>Class</strong>: all positions with the same data value</td>
</tr>
<tr>
<td>• <strong>Interval</strong>: (maximal set of) contiguous positions of a class</td>
</tr>
<tr>
<td>• <strong>String projection</strong> $P(s)$: $ cbcaabbbcabacbbaa $</td>
</tr>
</tbody>
</table>
**What $\text{FO}^2$ Can Express**

<table>
<thead>
<tr>
<th>Example properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>- Let $\alpha$ and $\beta$ denote unary types</td>
</tr>
<tr>
<td>- $\text{FO}^2$ can express</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>
What $\text{FO}^2$ Can Express

<table>
<thead>
<tr>
<th>Example properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Let $\alpha$ and $\beta$ denote unary types</td>
</tr>
<tr>
<td>● $\text{FO}^2$ can express</td>
</tr>
<tr>
<td>– <strong>data-blind</strong> properties, i.e., properties not using $\sim$</td>
</tr>
</tbody>
</table>


München May 05 Logic and XML 59 Thomas Schwentick
What $\text{FO}^2$ Can Express

Example properties

- Let $\alpha$ and $\beta$ denote unary types
- $\text{FO}^2$ can express
  - **data-blind** properties, i.e., properties not using $\sim$
  - All occurrences of a type $\alpha$ are in the same class:
    $$\theta = \forall x \forall y ((\alpha(x) \land \alpha(y)) \rightarrow x \sim y)$$
What $\text{FO}^2$ Can Express

<table>
<thead>
<tr>
<th>Example properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Let $\alpha$ and $\beta$ denote unary types</td>
</tr>
<tr>
<td>• $\text{FO}^2$ can express</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>• <strong>data-blind</strong> properties, i.e., properties not using $\sim$</td>
</tr>
<tr>
<td>• All occurrences of a type $\alpha$ are in the same class:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>• Each class contains at most one occurrence of $\alpha$:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
What $\text{FO}^2$ Can Express

Example properties

- Let $\alpha$ and $\beta$ denote unary types
- $\text{FO}^2$ can express
  - **data-blind** properties, i.e., properties not using $\sim$
  - All occurrences of a type $\alpha$ are in the same class:
    $$\theta = \forall x \forall y ((\alpha(x) \land \alpha(y)) \rightarrow x \sim y)$$
  - Each class contains at most one occurrence of $\alpha$:
    $$\theta = \forall x \forall y ((\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y)$$
  - In each class, every $\alpha$ occurs before every $\beta$:
    $$\theta = \forall x \forall y ((\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x < y)$$
Let $\alpha$ and $\beta$ denote unary types

\( \text{FO}^2 \) can express

- **data-blind** properties, i.e., properties not using $\sim$

- All occurrences of a type $\alpha$ are in the same class:
  \[
  \theta = \forall x \forall y \left( (\alpha(x) \land \alpha(y)) \rightarrow x \sim y \right)
  \]

- Each class contains at most one occurrence of $\alpha$:
  \[
  \theta = \forall x \forall y \left( (\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y \right)
  \]

- In each class, every $\alpha$ occurs before every $\beta$:
  \[
  \theta = \forall x \forall y \left( (\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x < y \right)
  \]

- Each class with an $\alpha$ also has a $\beta$:
  \[
  \theta = \forall x \exists y \left( \alpha(x) \rightarrow (\beta(y) \land x \sim y) \right)
  \]
What $FO^2$ Can Express

<table>
<thead>
<tr>
<th>Example properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Let $\alpha$ and $\beta$ denote unary types</td>
</tr>
<tr>
<td>● $FO^2$ can express</td>
</tr>
<tr>
<td>– data-blind properties, i.e., properties not using $\sim$</td>
</tr>
<tr>
<td>– All occurrences of a type $\alpha$ are in the same class:</td>
</tr>
<tr>
<td>$\theta = \forall x \forall y ((\alpha(x) \land \alpha(y)) \rightarrow x \sim y)$</td>
</tr>
<tr>
<td>– Each class contains at most one occurrence of $\alpha$:</td>
</tr>
<tr>
<td>$\theta = \forall x \forall y ((\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y)$</td>
</tr>
<tr>
<td>– In each class, every $\alpha$ occurs before every $\beta$:</td>
</tr>
<tr>
<td>$\theta = \forall x \forall y ((\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x &lt; y)$</td>
</tr>
<tr>
<td>– Each class with an $\alpha$ also has a $\beta$:</td>
</tr>
<tr>
<td>$\theta = \forall x \exists y (\alpha(x) \rightarrow (\beta(y) \land x \sim y))$</td>
</tr>
<tr>
<td>– It turns out: That’s basically all!</td>
</tr>
</tbody>
</table>
Proof Structure

Main steps of the proof

- $FO^2$ formula $\varphi$
  - Scott normal form
  - Intermediate normal form
  - Data normal form $\psi$

- Construct multicounter automaton $A_\psi$ such that:
  - $A_\psi$ accepts a string $w$ if and only if
  - there is a data string $s$ with
    * $s \models \varphi$
    * $P(s) = w$

- Check whether $L(A_\psi) \neq \emptyset$
Proof Structure

Main steps of the proof

- \( FO^2 \) formula \( \varphi \)
  \[
  \downarrow
  \]
  Scott normal form
  \[
  \downarrow
  \]
  Intermediate normal form
  \[
  \downarrow
  \]
  Data normal form \( \psi \)

- Construct multicounter automaton \( A_\psi \) such that:
  - \( A_\psi \) accepts a string \( w \) if and only if
  - there is a data string \( s \) with
    * \( s \models \varphi \)
    * \( P(s) = w \)

- Check whether \( L(A_\psi) \neq \emptyset \)

Definition: Multicounter-automaton

- Nondeterministic string automaton
  (not: data string!)
- Finite number of counters
- Counter values \( \geq 0 \) (or reject)
- No intermediate test whether counter is 0
- Acceptance if finally all counters are 0

Remarks

- Closely related to Petri nets
- Non-emptiness: decidable
  [Sacerdote, Tenney 77]
- Lower bound: \( \text{EXPSPACE} \)
- No elementary upper bound known
Normalization

Normal forms

● We transform into equivalent EMSO formulas

● **Scott normal form**: \( \exists R_1, \ldots, R_k \ \forall x \forall y \chi \land \land_i \forall x \exists y \chi_i \)

● **Intermediate normal form**: \( \exists R_1 \cdots R_m \theta_1 \land \cdots \land \theta_n \)

● \( \theta_i \):

(1) \( \forall x \forall y \ (\delta(x,y) \geq 2 \land \alpha(x) \land \beta(y) \land \begin{array}{l} x \sim y \\
 x \not\sim y \end{array}) \rightarrow \begin{array}{l} x < y \\
 x > y \end{array} \)

(2) \( \forall x \exists y \ \alpha(x) \rightarrow (\beta(y) \land \begin{array}{l} x \sim y \\
 x \not\sim y \end{array} \land \begin{array}{l} x + 1 < y \\
 x + 1 = y \\
 x = y \\
 x = y + 1 \\
 x > y + 1 \end{array}) \)

● Note: \( \forall \forall \) without +1
Normal forms (cont.)

- **Data normal form**: Disjunction of formulas
  \[ \exists R_1 \cdots R_n R_\# \theta_1 \land \cdots \land \theta_n \]

- \( \theta_i \):
  - (a) data-blind
  - (b) All \( \alpha \) are in the same class
  - (c) Each class contains at most one \( \alpha \)
  - (d) In each class, every \( \alpha \) occurs before every \( \beta \)
  - (e) Each class with an \( \alpha \) also has a \( \beta \)
  - (f) \( R_\# \) marks the first position of each interval:
    \[ \forall x R_\#(x) \leftrightarrow \forall y (x = y + 1 \rightarrow x \not\sim y) \]
### Normalization (cont.)

**Data normal form**: Disjunction of formulas

\[ \exists R_1 \cdots R_n R_\# \; \theta_1 \land \cdots \land \theta_n \]

**\( \theta_i \)**:

(a) data-blind

(b) All \( \alpha \) are in the same class

(c) Each class contains at most one \( \alpha \)

(d) In each class, every \( \alpha \) occurs before every \( \beta \)

(e) Each class with an \( \alpha \) also has a \( \beta \)

(f) \( R_\# \) marks the first position of each interval:

\[ \forall x R_\#(x) \leftrightarrow \forall y(x = y + 1 \rightarrow x \not< y) \]

### Normalization steps

**\( \text{FO}^2 \rightarrow \text{Scott normal form} \): standard**

**Scott normal form**

\[ \rightarrow \text{intermediate normal form}: \]
relatively straightforward

**Intermediate normal form**

\[ \rightarrow \text{data normal form}: \]

- For each type \( \alpha \) we capture the two left-most classes with \( \alpha \) and the two rightmost classes with \( \alpha \) by unary relations \( R_1^\alpha, \ldots, R_4^\alpha \)

- Case distinction on possible formulas (1) and (2)

\[ \rightarrow \text{in each case } \theta_i \text{ can be replaced by some “data normal” formulas} \]
Recall ingredients of data normal form:
(a) data-blind
(b) All $\alpha$ are in the same class
(c) Each class contains at most one $\alpha$
(d) In each class, every $\alpha$ occurs before every $\beta$
(e) Each class with an $\alpha$ also has a $\beta$
(f) $R\# \in \mathbb{N}$ marks the first position of each interval:
$$\forall x R\#(x) \leftrightarrow \forall y(x = y + 1 \rightarrow x \not< y)$$

- (a), (f): straightforward
- (c), (d), (e) induce regular conditions for each class: $L$
- (b) specifies regular conditions for some special classes: $L_1, \ldots, L_k$
- Multicounter automaton $\mathcal{A}$ accepts basically shuffle of $L, L_1, \ldots, L_k$
Construction of Multicounter automaton (cont.)

Proof (cont.)

- \( A \) accepts string projections of models of disjunctions of formulas
  \[ \exists R_1 \cdots R_n R# \theta_1 \land \cdots \land \theta_n \]
- \( A \) guesses a disjunct
- \( A \) guesses, for each position \( R_1, \ldots, R_n, R# \)

\[ \rightarrow \] In particular: guesses intervals
- But: \( A \) does not know which intervals belong to the same class

- Special classes \( (L_1, \ldots, L_k) \) can be checked directly
  (if \( \alpha \) occurs in only one class this class is \( R_1^\alpha \))

- To check that all other class strings are in \( L \):
  - Let \( B \) be a string automaton for \( L \)
  - \( A \) has one counter per state of \( B \)
  - Counter \( C_q \) counts how many (non-special) class strings seen so far led to a state \( q \)
  - Complication: At interval border \( A \) can proceed from a state \( q \) that was just “reached” only if \( C_q \geq 2 \)
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
</tr>
<tr>
<td>MSO Logics</td>
</tr>
<tr>
<td>Weaker Logics</td>
</tr>
<tr>
<td>Extensions</td>
</tr>
<tr>
<td><strong>Conclusion</strong></td>
</tr>
</tbody>
</table>
Conclusion

What we have seen

Logic is useful for the theory of XML languages:

- MSO offers framework for schema languages
- MSO $\equiv$ regular node selecting queries
- Two-variable logic $\equiv$ XPath
- FO-logic $\equiv$ natural extension of XPath
- MSO helpful in the context of transformations
- Two-variable logic with data is decidable

Open

There remains a lot to be done, e.g.

- XQuery
- Automata in the presence of data values
- Practical relevance of logic-automata approach?
Conclusion

What we have seen

Logic is useful for the theory of XML languages:

- MSO offers framework for schema languages
- MSO \( \equiv \) regular node selecting queries
- Two-variable logic \( \equiv \) XPath
- FO-logic \( \equiv \) natural extension of XPath
- MSO helpful in the context of transformations
- Two-variable logic with data is decidable

Open

There remains a lot to be done, e.g.

- XQuery
- Automata in the presence of data values
- Practical relevance of logic-automata approach?

Finally

Thank You!