XML Graphs in Program Analysis

Anders Møller

Michael I. Schwartzbach

University of Aarhus

BRICS

Overview

- What are XML graphs
- Applications:
  - XACT
  - Java Servlets and JSP
  - XSugar
  - XSLT

Four applications

- **XACT** – Java-based transformations of XML fragments
  ⇒ static type-checking with XML Schema
- **Java Servlets and JSP**
  ⇒ static validation of output
- **XSugar** – dual syntax for XML languages
  ⇒ static checking of grammars vs. schemas
- **XSLT**
  ⇒ static validation of stylesheets,
  dead code detection

Main publications about XML graphs and XSLT analysis

- Anders Møller and Michael I. Schwartzbach, ***XML Graphs in Program Analysis***, invited paper at PEPM’07
- Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach, ***Static Validation of XSL Transformations***, to appear in TOPLAS 29(4)

Representing XML abstractions

- We need a versatile model of **sets of XML documents**

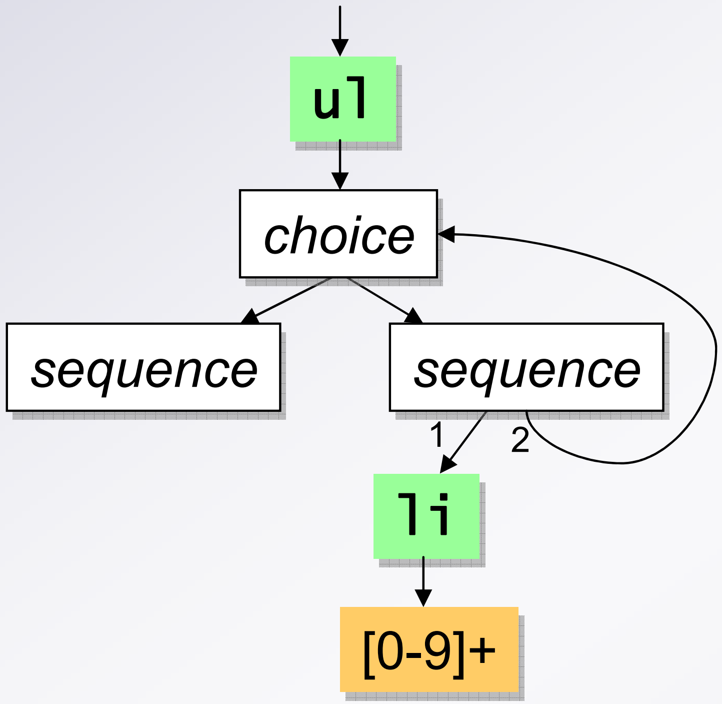
- Requirements:
  1. Capture **all of XML**, not an idealized subset
  2. Represent sets of XML documents described by **common formalisms** such as DTD and XML Schema
  3. Allow **static validation** against schemas
  4. Allow **static navigation** with XPath expressions
  5. Provide **finite-height lattice** structures for dataflow analysis and fixed-point iteration
  6. Be fully **implemented**

From XML trees to XML graphs

- **XML graphs** generalize XML trees:
  - Character data, attributes values, and element names are described by **regular string languages**
  - Not only sequence nodes for content, but also **choice** and **interleave** nodes
  - **Loops** are permitted
  - Special **gaps** to model XML fragments
- An XML graph represents **a set of XML templates**

– A pragmatic model fine-tuned through 6 years of program analysis development

Example of an XML graph



All `ul` lists
with zero or more `li` items
each containing a numeral

Formal definition of XML graphs

$\mathcal{X} = (\mathcal{N}, \mathcal{R}, \mathit{contents}, \mathit{strings}, \mathit{gaps})$

- $\mathcal{N}$ contains **nodes**
  (element, attribute, text, sequence, choice, interleave, gap)
- $\mathcal{R}$ is a subset of **root nodes**
- *contents* describe the **edges**
  (depending on the node kind)
- *strings* assigns **sets of strings** to certain nodes
  (element/attribute names, character data, attribute values)
- *gaps* describe information about **gaps**
  (only used in some applications, in particular XACT)

Unfolding semantics

$$n \Rightarrow x \,;\, t \,;\, a$$

a node — XML content — text — attributes

$$\mathcal{L}(\chi) = \{x \mid \exists n \in \mathcal{R} : n \Rightarrow x \,;\, t \,;\, a\}$$

Unfolding semantics

$$\frac{n \in N_{\mathcal{E}} \quad s \in strings(n) \quad contents(n) \Rightarrow x \,;\, t \,;\, a}{n \Rightarrow \texttt{<}s\ a\texttt{>}\ x\ \texttt{</}s\texttt{>} \,;\, \varepsilon \,;\, \emptyset} \text{[element]}$$

$$\frac{n \in N_{\mathcal{A}} \quad s \in strings(n) \quad contents(n) \Rightarrow x \,;\, t \,;\, a \quad t \neq \emptyset}{n \Rightarrow \varepsilon \,;\, \varepsilon \,;\, s\texttt{="}t\texttt{"}} \text{[attribute]}$$

$$\frac{n \in N_{\mathcal{T}} \quad s \in strings(n)}{n \Rightarrow s \,;\, s \,;\, \emptyset} \text{[text]}$$

$$\frac{\begin{array}{c} n \in N_{\mathcal{S}} \quad contents(n) = m_1 \cdots m_k \\ m_i \Rightarrow x_i \,;\, t_i \,;\, a_i \qquad a \in a_1 \oplus \cdots \oplus a_k \end{array}}{n \Rightarrow x_1 \cdots x_k \,;\, t_1 \cdots t_k \,;\, a} \text{[sequence]}$$

$$\frac{n \in N_{\mathcal{C}} \cup N_{\mathcal{G}} \quad m \in contents(n) \quad m \Rightarrow x \,;\, t \,;\, a}{n \Rightarrow x \,;\, t \,;\, a} \text{[choice]}$$

Lattice structure

- XML graphs are **compatible** if they differ only on
  - *roots*
  - *strings*
  - *choice-node edges*, and
  - *gaps*

  (i.e. they agree on the nodes and the non-choice-node edges)

- Compatible XML graphs are ordered pointwise
  - they form a **lattice**!
    (finite-height if *strings* has finite co-domain)

- Non-compatible expansion is **polyvariance**

Operations on XML graphs

- XML **documents** are a special case
- DTD, XML Schema, and RELAX NG can be **represented exactly**
- Closed under **union** and **least upper bound** (on compatible graphs)
- Closed under **gap/template plugging**
- **Validation** relative to a given schema is possible
- **XPath** location paths can be evaluated

Relations to other formalisms

- Theoretically quite close to:
  - RELAX NG
  - regular tree grammars
  - regular expression types (XDuce types)

- Pragmatic advantages:
  - Lattice structure
  - Includes text, attributes, and interleaving
  - Some non-regular structures can be expressed
  - Maintains template gap information

Implementation

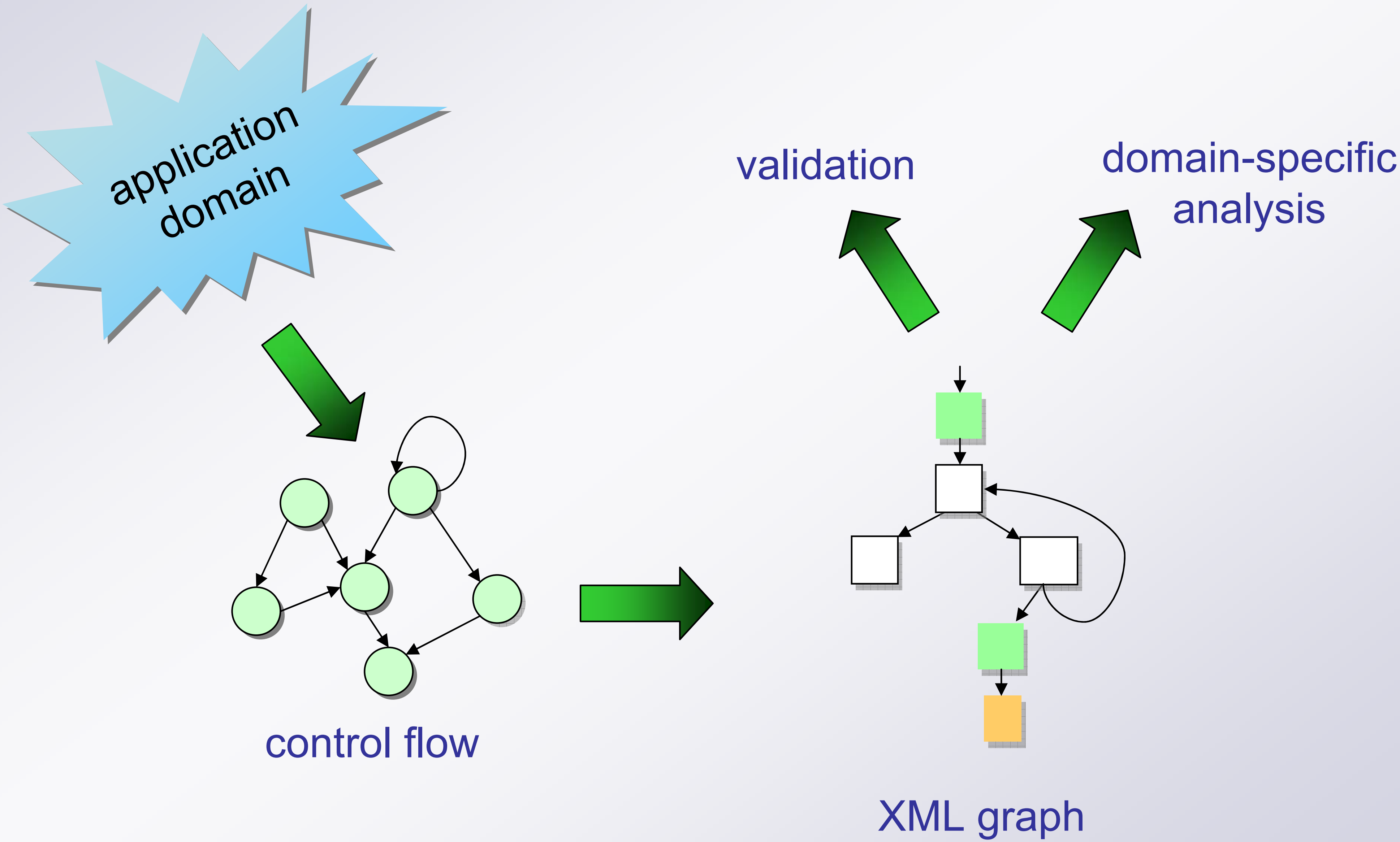
- Open source Java library: `dk.brics.schematools`

- **Representation** of XML graphs
- Conversion from **XML documents** and **templates**
- Conversion from **schemas**, including XML Schema, to XML graphs and Restricted RELAX NG (essentially single-type tree grammars)
- **Validation** relative to XML Schema and Restricted RELAX NG schemas
- Evaluation of **XPath** location paths
- Command-line interface, as supplement to the API

Overview

- What are XML graphs
- **Applications:**
  - **XACT**
  - **Java Servlets and JSP**
  - **XSugar**
  - **XSLT**

Typical approach



Overview of XSLT analysis

- Brief summary of XSLT (1.0)
- Stylesheet mining
- Type checking XSLT stylesheets
  - simplification
  - flow analysis
  - XML graph construction and validation

XSLT 1.0

- XSLT (XSL Transformations) is designed for transformations for document-centric XML languages
- A *declarative domain-specific* language based on **templates** and **pattern matching** using XPath
- An XSLT program consists of **template rules**, each having a **pattern** and a **template**

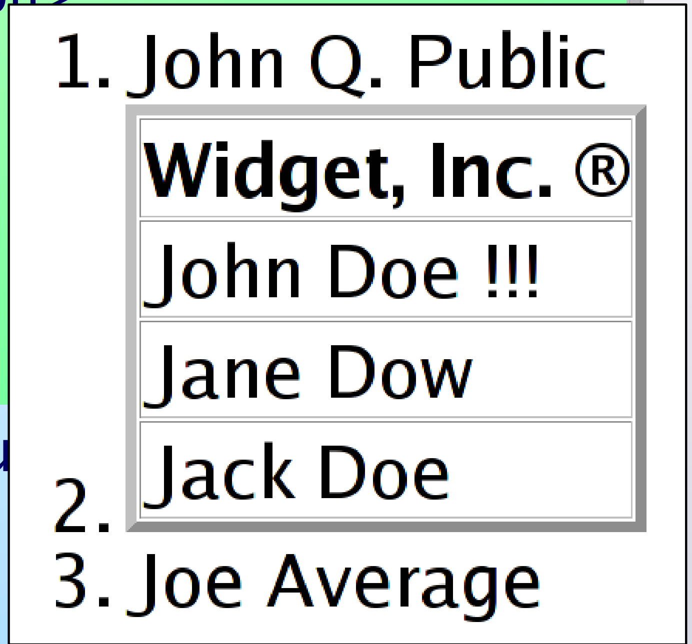
Processing model

- A **source XML tree** is transformed by processing its root node
- A single **node** is processed by
  - **finding** the template rule with the best matching pattern
  - **instantiating** its template
    - may create result fragments
    - may select other nodes for processing
- A **node list** is processed by processing each node and concatenating the results

An example input XML document

```
<registrations xmlns="http://eventsRus.org/registrations/">
  <name id="117">John Q. Public</name>
  <group type="private" leader="214">
    <affiliation>Widget, Inc.</affiliation>
    <name id="214">John Doe</name>
    <name id="215">Jane Dow</name>
    <name id="321">Jack Doe</name>
  </group>
  <name id="742">Joe Average</name>
</r
```

```
<!ELEMENT registrations (name|grou
<!ELEMENT name (#PCDATA)>
<!ATTLIST name id ID #REQUIRED>
<!ELEMENT group (affiliation,name*)>
<!ATTLIST group type (private|government) #REQUIRED
                leader IDREF #REQUIRED>
<!ELEMENT affiliation (#PCDATA)>
```



An XSLT stylesheet (1/3)

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:reg="http://eventsRus.org/registrations/"
                xmlns="http://www.w3.org/1999/xhtml">

   <xsl:template match="reg:registrations">
     <html>
       <head><title>Registrations</title></head>
       <body>
         <ol><xsl:apply-templates/></ol>
       </body>
     </html>
   </xsl:template>

   <xsl:template match="*">
     <li><xsl:value-of select="."/></li>
   </xsl:template>
```

An XSLT stylesheet (2/3)

```
<xsl:template match="reg:group">
  <li>
    <table border="1">
      <thead>
        <tr>
          <td>
            <xsl:value-of select="reg:affiliation"/>
            <xsl:if test="@type='private'">&#174;</xsl:if>
          </td>
        </tr>
      </thead>
      <xsl:apply-templates select="reg:name">
        <xsl:with-param name="leader" select="@leader"/>
      </xsl:apply-templates>
    </table>
  </li>
</xsl:template>
```

An XSLT stylesheet (3/3)

```
  <xsl:template match="reg:group/reg:name">
    <xsl:param name="leader" select="-1"/>
    <tr>
      <td>
        <xsl:value-of select="."/>
        <xsl:if test="$leader=@id">!!!</xsl:if>
      </td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

Templates

Main template constructs:

- *literal result fragments*
  - character data, non-XSLT elements
- *recursive processing*
  - `apply-templates`, `call-template`, `for-each`, `copy`, `copy-of`
- *computed result fragments*
  - `element`, `attribute`, `value-of`, ...
- *conditional processing*
  - `if`, `choose`
- *variables and parameters*
  - `variable`, `param`, `with-param`

use **XPath** for computing values

The challenge

Given

- an XSLT stylesheet S, and
- two schemas, D_{in} and D_{out}

assuming that X is valid relative to D_{in}

is S applied to X always valid relative to D_{out}?

– undecidable, we aim for a **conservative approximation**

Overview of XSLT analysis

- Brief summary of XSLT (1.0)
- **Stylesheet mining**
- Type checking XSLT stylesheets
  - simplification
  - flow analysis
  - XML graph construction and validation

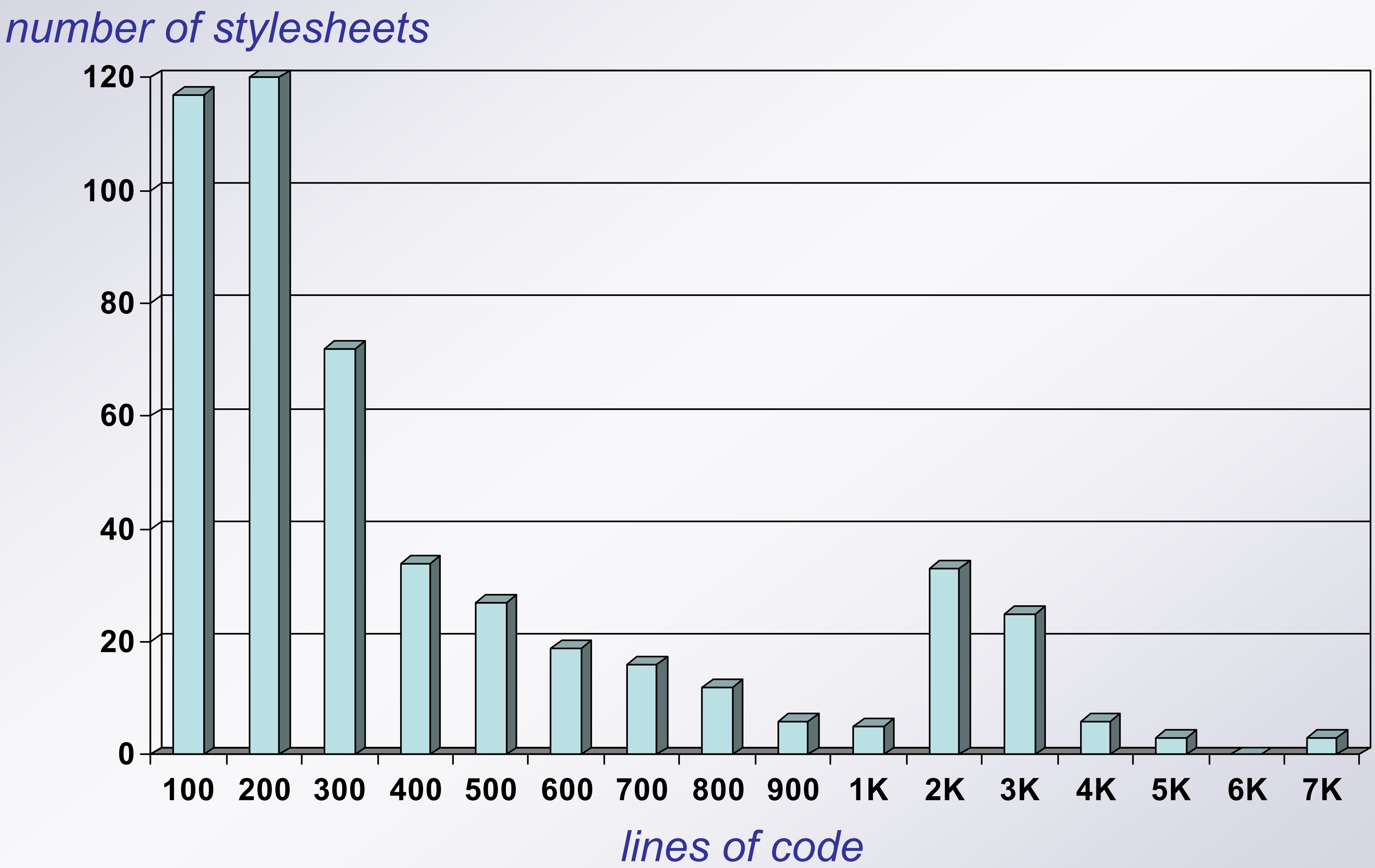
Stylesheet mining

- XSLT is a big language...

- How are the many features of XSLT being used?
  - typical stylesheet size?
  - complexity of `select` expressions?
  - complexity of `match` expressions?

- Obtained via Google:
  499 stylesheets with a total of
  186,726 lines of code

Stylesheet sizes



Complexity of `select` expressions

Category	Number	Fraction
default	3,415	31.2%
`a`	3,335	30.4%
`a/b/c`	1,153	10.5%
`*`	740	6.8%
`a	b	c`	473	4.3%
`text()`	235	2.1%
`a[...]`	223	2.0%
`/a/b/c`	110	1.0%
`a[...]/b[...]/c[...]`	82	0.7%
`@a`	68	0.6%
...	...	...
name(s) known	602	5.6%
nasty	175	1.6%
Total	10,768	100.0%

Complexity of `match` expressions

Category	Number	Fraction
`a`	**4,710**	**53.9%**
absent	**1,369**	**15.7%**
`a/b`	**523**	**6.0%**
`a[@b='…']`	**467**	**5.3%**
`a/b/c`	**423**	**4.8%**
`/`	**256**	**2.9%**
`*`	**217**	**2.5%**
`a | b | c`	**177**	**2.0%**
…	…	…
`a[…]`	**225**	**2.6%**
`…/a[…]`	**225**	**2.6%**
`…/a`	**108**	**1.2%**
…	…	…
nasty	**97**	**1.1%**
Total	**8,739**	**100.0%**

Overview of XSLT analysis

- Brief summary of XSLT (1.0)
- Stylesheet mining
- **Type checking XSLT stylesheets**
  - simplification
  - flow analysis
  - XML graph construction and validation

The XSLT validation algorithm

Our strategy:

1. reduce *S* to **core** features of XSLT
2. analyze **flow** (using D_{in})
   - `apply-templates` $\rightarrow$ `template` ?
   - possible context nodes when templates are instantiated?
3. construct **XML graph**
4. **validate** XML graph relative to D_{out}

Semantics preserving simplifications

- make **defaults** explicit (built-in template rules, default `select`, default axes, coercions, ...)
- insert **`imported`**/**`included`** stylesheets
- convert **literal** elements and attributes to `element`/`attribute` instructions
- convert **text** to `text` instructions
- expand **variable uses** (not parameters)
- reduce **`if`** to `choose`
- reduce **`for-each`**, **`call-template`**, and **`copy`** to `apply-templates` instructions and new template rules
- move **nested templates** (in `when`/`otherwise`) to new template rules

Approximating simplifications

- replace each **`number`** by a `value-of` with `xslv:unknownString()`
- replace each **`value-of`** expression by `xslv:unknownString()`, except for `string(self::node())` and `string(attribute::a)`
- replace **`when`** conditions by `xslv:unknownBoolean()`
- replace **`name`** attributes in **`attribute`** and **`element`** instructions by `{xslv:unknownString()}`, except for constants and `{name()}`

Reduced XSLT

The only features left:

- **`template`** rules with `match`, `priority`, `mode`, `param`
- **`apply-templates`** with `select`, `mode`, `sort`, `with-param`
- **`choose`** where each condition is `xslv:unknownBoolean()` and each branch template is an `apply-templates`
- **`copy-of`** with a parameter as argument
- **`attribute`** and **`element`** whose name is a constant, `{name()}` or `{xslv:unknownString()}` and the contents of `attribute` is a `value-of`
- **`value-of`** where the argument is `xslv:unknownString()`, `string(self::node())` or `string(attribute::a)`
- top-level **`param`** declarations (no variables)

– and that's all!

Overview of XSLT analysis

- Brief summary of XSLT (1.0)
- Stylesheet mining
- Type checking XSLT stylesheets
  - simplification
  - **flow analysis**
  - XML graph construction and validation

Flow analysis

Goals:

- Determine **flow** from `apply-templates` nodes to `template` nodes
- Determine possible **context nodes** for instantiated `template` nodes

Flow graphs

- Define
  - $\Sigma = \mathcal{E} \cup (\mathcal{A} \times \mathcal{E}) \cup \{root, pcdata, comment, pi\}$ (describes types of possible context nodes)
  - A_s = `apply-templates` nodes for S
  - T_s = `template` nodes for S

- A **flow graph** is a pair $G = (C, F)$ where
  - $C : T_s \rightarrow 2^\Sigma$ describes the *context sets*
  - $F : A_s \times T_s \rightarrow (\Sigma \rightarrow 2^\Sigma)$ describes the *edge flow*

A typical situation

t

```
<template match="match_t">
   .
   .
a  <apply-templates select="select_a"/>
   .
   .
</template>
```

{ σ }

$\Phi(\sigma, select_{a}, match_{t'}, match_{t})$

t'

```
<template match="match_t'">
   .
   .
</template>
```

Fixed point algorithm

Find smallest solution to these **constraints**:

- $root \in C(t)$
  if the `match` expression of t matches the root

- $\sigma \in C(t) \Rightarrow \Phi(\sigma, select_a, match_{t'}, match_t) \subseteq F(a,t')(\sigma)$
  where $\Phi(\ldots)$ is an upper approximation of the possible flow from a in t to t' starting with σ

- $F(a,t)(\sigma) \subseteq C(t)$

How to compute Φ ???

- ***match*** expressions are always **downward**
- According to our stylesheet mining, most ***select*** expressions are also downward!
  - and the rest can be approximated by downward expressions

Define **regular languages**:

R(x) = strings over Σ corresponding to downward XPath location path x

$\Pi(D)$ = strings over Σ corresponding to downwards paths allowed by schema D

Computing Φ with downward paths

A good version of Φ is computed using finite-state automata:

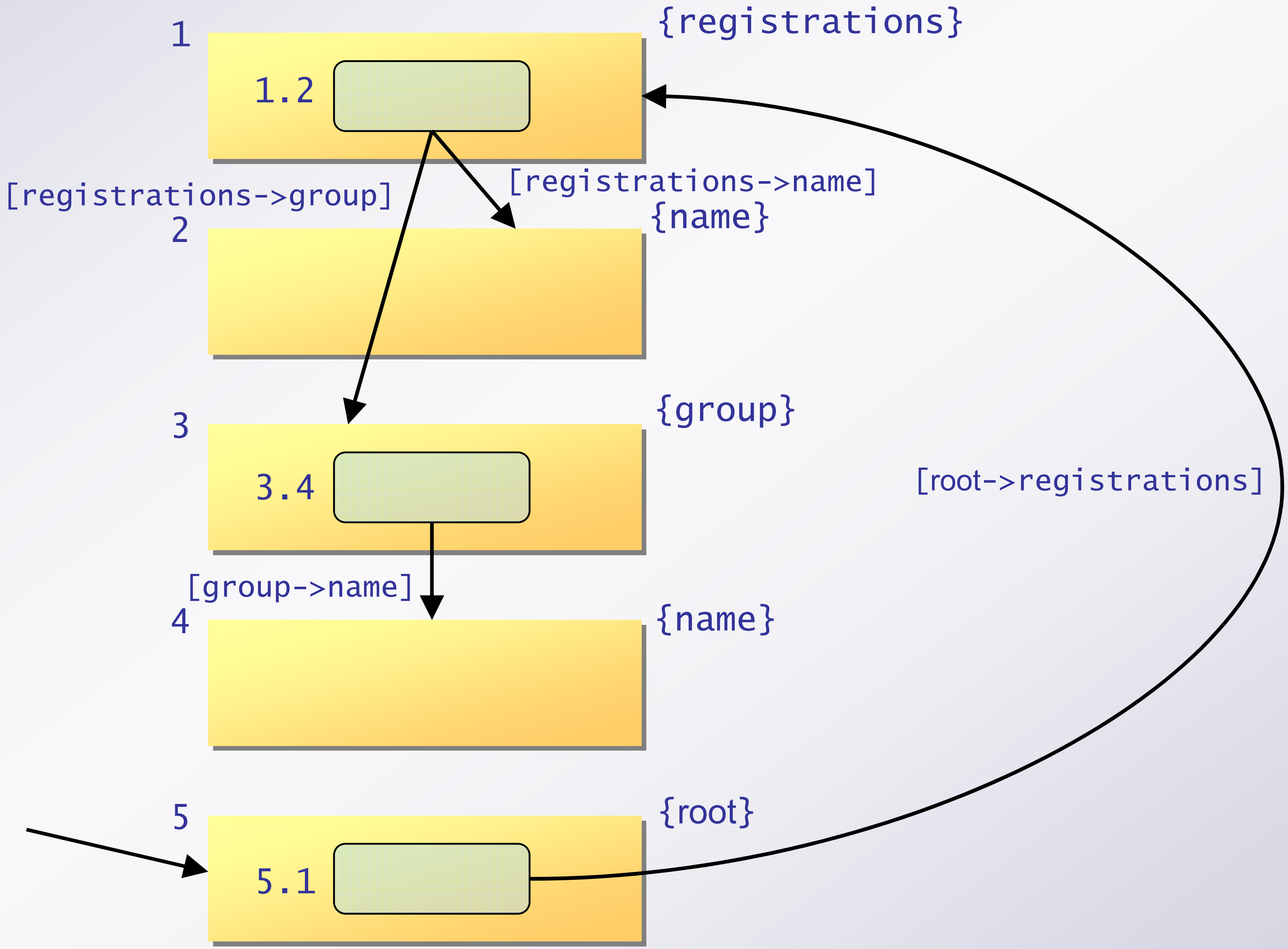
$$\sigma' \in \Phi(\sigma, \mathit{select}_a, \mathit{match}_{t'}, \mathit{match}_t)$$

iff

$$\omega\sigma' \in \Sigma^* R(\alpha) \cap \Sigma^* R(\mathit{match}_{t'}) \cap \Pi(D_{in})$$

where $\alpha = \begin{cases} \mathit{select}_a & \text{if } \mathit{select}_a \text{ starts with } / \\ \mathit{match}_t \,/\, \mathit{type}(\sigma) \,/\, \mathit{select}_a & \text{otherwise} \end{cases}$

Example flow graph



Overview of XSLT analysis

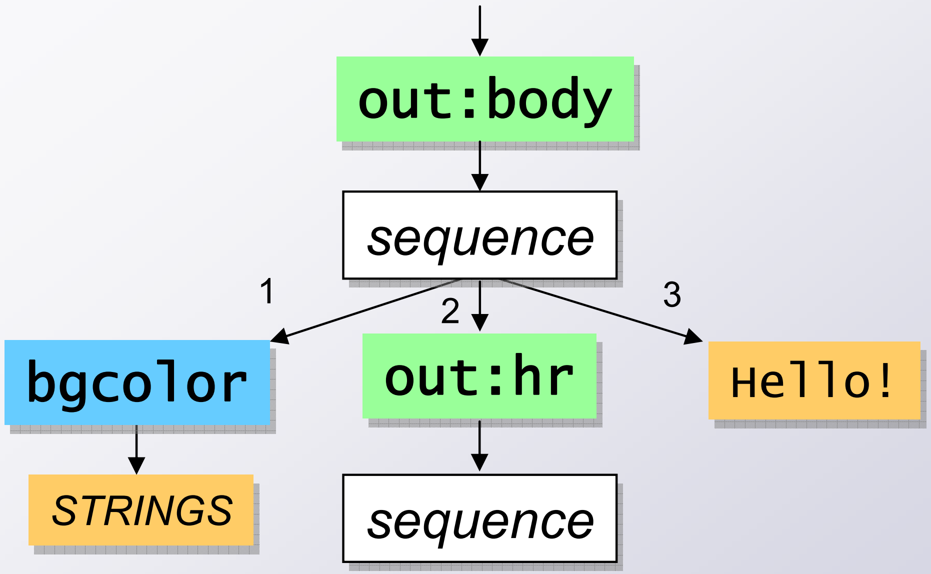
- Brief summary of XSLT (1.0)
- Stylesheet mining
- Type checking XSLT stylesheets
  - simplification
  - flow analysis
  - **XML graph construction and validation**

XML graph fragments

- For each **template** $t \in T_s$ and $\sigma \in C(t)$ we construct an **XML graph fragment** describing the possible XML output
  - the fragment has *placeholders* for occurrences of `apply-templates` nodes
  - the construction is performed recursively in the template structure (lots of special cases)

A fragment example

```
<element name="out:body">
  <attribute name="bgcolor">
    <value-of select="xslv:unknownString()"/>
  </attribute>
  <element name="out:hr"/>
  <value-of select="'Hello!'"/>
</element>
```



Connecting fragments

- The fragments for templates are connected using
  - the `select` attributes in `apply-templates` nodes
  - the information in the flow graph
  - the information in the input schema

- *The challenge is to capture the content model of the output language with sufficient precision*

XML graph validation

- We now have an **XML graph** that conservatively models the output language
- We must check that its language is accepted by the output schema D_{out}
- This can be done using `dk.brics.schematools` ☺

Validation errors

A typical error message:

```
*** Validation error
Source: element {http://www.w3.org/1999/xhtml}pre at
  list2html line 120 column 25
Schema: xhtml1-transitional.rng line 1284 column 25
Error: invalid child:
  {http://www.w3.org/1999/xhtml}map
```

Related work

- Audebaud & Rose 2000:
  - typing rules
  - tiny fragment of XSLT
- Tozawa 2001:
  - inverse type inference (Milo, Suciu, Vianu)
  - even smaller fragment, not implemented
- Dong & Bailey 2004:
  - coarser (but cheaper) flow analysis
  - used for debugging (not static validation)

Recent work

- Full **XSLT 2.0** (and thus full XPath 2.0)
- Full **XML Schema**, not just DTD
- Much faster than our first implementation:
  - weeds out potential flow edges with Dong & Bailey's technique
  - avoids expensive automata computations without loss of precision
- Online demo: http://www.brics.dk/XSLV

Conclusion

- **XML graphs** are useful for representing sets of XML documents in program analysis
- Example application:
  practical validity analyzer for ***XSLT***

  Methodology:
  - **mining** to learn about XSLT in practice
  - reduce to **core features**
  - **pragmatic, conservative approximation**
  - **flow analysis** (`apply-templates` → `template`)
  - **XML graphs** for validation