Static Validation of XSL Transformations

Mads Østerby Olesen
Søren Kuula
Anders Møller
Michael Schwartzbach

University of Aarhus
BRICS

Plan

- 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 **stylesheet** 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|group)*>
<!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)

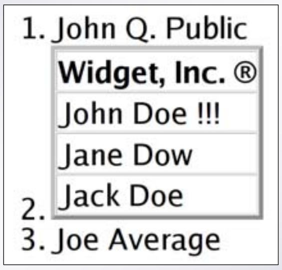
```
<xsl:template match="reg:group">
  <li>
    <table border="1">
      <thead>
        <tr>
          <td>
            <xsl:value-of select="reg:affiliation"/>
            <xsl:if test="@type='private'">&#174;</xsl:if>
          </td>
        </tr>
      </thead>
      <xsl:apply-templates select="reg:name">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


The resulting output



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*,
- 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} ?

Undecidability

- Since XSLT is **Turing complete**, this question is clearly **undecidable**
- Thus, we must apply some **approximations**
- We want to be able to **guarantee correctness**
- Thus, we go for ***conservative*** *approximations* (i.e. err on the safe side)

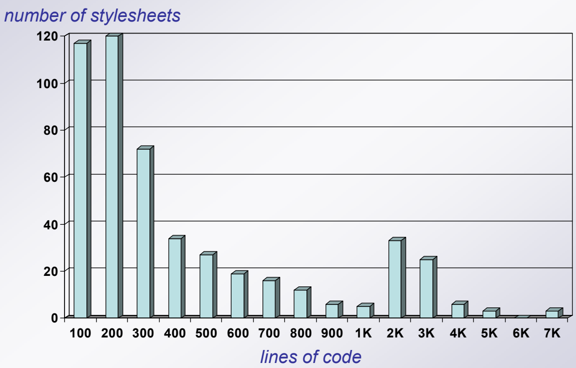
Plan

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

Stylesheet mining

- 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%
/a[...]/b[...]/c[...]	43	0.4%
..	32	0.3%
/	8	0.1%
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%
text()	52	0.6%
@a	24	0.3%
@*	16	0.2%
a:*	12	0.1%
processing-instruction()	11	0.1%
@n:*	4	0.0%
a[...]	225	2.6%
.../a[...]	225	2.6%
.../a	108	1.2%
.../@a	24	2.7%
.../text()	11	0.1%
.../n:*	1	0.0%
nasty	97	1.1%
Total	8,739	100.0%

Plan

- 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` → `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 `import`ed/`include`d 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()}`

(We want to handle ***almost-identity* transformations**)

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)

The reduced XSLT stylesheet (1/4)

```
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
            xmlns:xslv="http://www.brics.dk/XSLTValidator"
            xmlns:in="http:///eventsRus.org/registrations"
            xmlns:out="http://www.w3.org/1999/xhtml"
            version="1.0">

  <!-- 1 -->
  <template match="in:registrations" priority="0">
    <element name="out:html">
      <element name="out:head">
        <element name="out:title">
          <value-of select="'Registrations'"/><!-- 1.1 -->
        </element>
      </element>
      <element name="out:body">
        <element name="out:ol">
          <apply-templates select="child::node()"/><!-- 1.2 -->
        </element>
      </element>
    </element>
  </template>
```


The reduced XSLT stylesheet (2/4)

```
<!-- 2 -->
<template match="*" priority="-0.5">
  <element name="out:li">
    <value-of select="string(self::node())"/><!-- 2.1 -->
  </element>
</template>
```

The reduced XSLT stylesheet (3/4)

```
<!-- 3 -->
<template match="in:group" priority="0">
  <element name="out:li">
    <element name="out:table">
      <attribute name="border" select="'1'"/><!-- 3.1 -->
      <element name="out:thead">
        <element name="out:tr">
          <element name="out:td">
            <value-of select="in:affiliation"/><!-- 3.2 -->
            <choose>
               <when test="xslv:unknownBoolean()">
                 <value-of select="'&#174;'"/><!-- 3.3 -->
               </when>
               <otherwise/>
            </choose>
          </element>
        </element>
      </element>
      <apply-templates select="in:name"><!-- 3.4 -->
        <with-param name="leader" select="@leader"/>
      </apply-templates>
    </element>
  </element>
</template>
```


The reduced XSLT stylesheet (4/4)

```
<!-- 4 -->
<template match="in:group/in:name" priority="0.5">
  <param name="leader" select="-1"/>
  <element name="out:tr">
    <element name="out:td">
      <value-of select="string(self::node())"/><!-- 4.1 -->
      <choose>
        <when test="xslv:unknownBoolean()">
          <value-of select="'!!!'"/><!-- 4.2 -->
        </when>
        <otherwise/>
      </choose>
    </element>
  </element>
</template>

<!-- 5 -->
<template match="/" priority="-1">
  <apply-templates select="child::node()"/><!-- 5.1 -->
</template>
</stylesheet>
```


Plan

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

Flow analysis

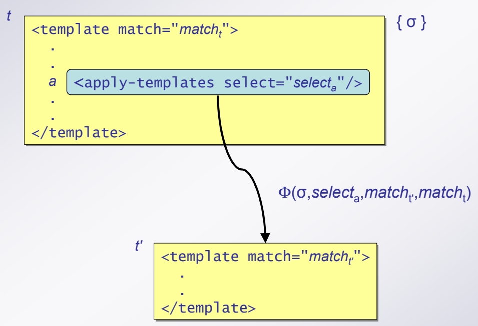
Goals:

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

Flow graphs

- Define
  - $\Sigma = \mathcal{E} \cup (\mathcal{A} \times \mathcal{E}) \cup \{root, pcdata, comment, pi\}$
  - 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



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

Computing Φ

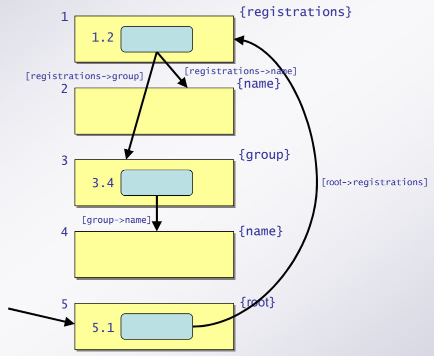
A good version of Φ is computed using DFAs:

$\sigma' \in \Phi(\sigma, select_a, match_t, match_{t'})$
iff
$\omega\sigma' \in \Sigma^* R(\alpha) \cap \Sigma^* R(match_{t'}) \cap \Pi(D_{in})$

$$\text{where } \alpha = \begin{cases} select_a & \text{if } select_a \text{ starts with } / \\ match_t \,/\, type(\sigma) \,/\, select_a & \text{otherwise} \end{cases}$$

$R(x)$ = downwards paths to targets of x
$\Pi(D)$ = downwards paths allowed by D

Example flow graph



Refinements

- Handling **modes**:
  – *delete flow edges with mismatching modes*
- Handling **priorities**:
  – *delete flow edges overshadowed by flow edges to higher priority templates*
- Handling **predicates**:
  – *use a simple theorem prover (not done yet)*

Plan

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

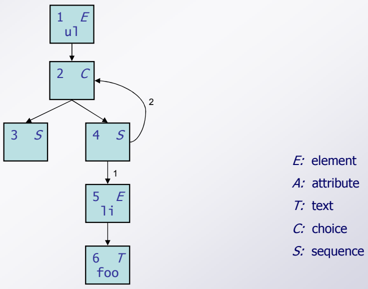
XML graphs

- A representation for sets of XML trees
- Different kinds of nodes:
  - element nodes
  - attribute nodes
  - text nodes

  (element, attribute and text nodes: names and values described by regular string languages)
  - sequence nodes
  - choice nodes
- The language of an XML graph is the set of XML trees obtained by finite unfoldings

Example XML graph

A graph for `ul` lists with zero or more `li` items:

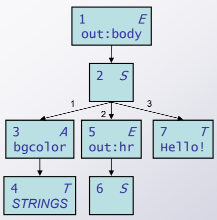


XML graph fragments

- 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"/>
  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*

Default expressions (31.7%)

It selects all child nodes of the context node σ

- Look up the **content model** of σ in D_{in}
- Construct an XML graph fragment for this regular language with placeholders for the input elements
- Replace each placeholder γ with a choice node leading to the fragments for each template t' such that $\gamma \in F(a,t')(\sigma)$

Projected contents (44.8%)

Cases such as
- `a`
- `*`
- `a | b | c`

- Proceed as for the default case
- But **project** the XML graph fragment onto the chosen sub-alphabet

Multiple location steps (11.9%)

Cases such as

- `a/b/c`
- `/a/b/c`

- One step at a time, **concatenate** the fragments

Predicates (3.4%)

Cases such as

- `a[...]/b[...]/c[...]`
- `/a[...]/b[...]/c[...]`

- First ignore the predicates
- Then add an edge to an empty sequence to model the case of a false predicate

Attributes (0.6%)

The case `@a`

- Use D_{in} to determine whether a is optional

Parent and root (0.4%)

Cases such as:

- `..`
- `/`

- A single node is selected
- Use a choice node with edges to the corresponding outgoing flow

Others (7.2%)

- Use a pessimistic content model containing all sequences of the outgoing flow
- If the possible names are known (5.6%), then use a cardinality analysis (0,1,*,+) for more precision

Sorting

- If the `apply-templates` contains `sort` directives, then scramble the determined content model

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 is done using an algorithm from the JWIG and XACT projects ☺

Validation errors

A typical error message:

```
*** Validation error:
     contents of element 'item' does not match declaration
    Rule:         <template match="child::*">...</template>
    Context node: item
    Element:      <item category="national">...</item>
    Schema:       (headline,text)
```


Benchmarks

Stylesheet	Lines	Input Schema	Lines	Output Schema	Lines
poem.xsl	35	poem.dtd	8	xhtml.dsd	2,278
AffordableSupplies.xsl	42	Catalog.dtd	31	xhtml.dsd	2,278
agenda.xsl	43	agenda.dtd	19	xhtml.dsd	2,278
news.xsl	54	news.dtd	12	xhtml.dsd	2,278
CreateInvoice.xsl	74	PurchaceOrder.dtd	37	dtdgen.dtd	32
adressebog.xsl	76	dtdgen.dtd	22	xhtml.dsd	2,278
order.xsl	112	order.dtd	31	fo.dtd	1,480
slideshow.xsl	118	slides.dtd	26	xhtml.dtd	1,198
psicode-links.xsl	145	links.dtd	15	xhtml.dtd	1,198
ontopia2xtm.xsl	188	tmstrict.dtd	113	xtm.dtd	202
proc-def.xsl	247	proc.dtd	69	xhtml.dtd	1,198
email_list.xsl	257	dtdgen.dtd	41	xhtml.dtd	1,198
tip.xsl	262	dtdgen.dtd	56	xhtml.dsd	2,278
window.xsl	701	dtdgen.dtd	84	xhtml.dtd	1,198
dsd2-html.xsl	1,353	dsd2.dtd	104	xhtml.dsd	2,278

Precision

Stylesheet	True Errors	False Errors
poem.xsl	2	0
AffordableSupplies.xsl	2	0
agenda.xsl	2	0
news.xsl	0	0
CreateInvoice.xsl	4	2
adressebog.xsl	2	0
order.xsl	0	0
slideshow.xsl	12	0
psicode-links.xsl	20	0
ontopia2xtm.xsl	0	1
proc-def.xsl	6	0
email_list.xsl	3	0
tip.xsl	1	0
window.xsl	0	0
dsd2-html.xsl	0	0

Error examples

Six unique errors for `slideshow.xsl`:

```
***Validation error: contents of element 'ul' may not match declaration
***Validation error: required attribute missing in element 'img'
***Validation error: required attribute missing in element 'script'
***Validation error: sub-element 'div' of element 'p' not declared
***Validation error: sub-element 'html' of element 'div' not declared
***Validation error: sub-element 'li' of element 'div' not declared
```


Efficiency

Stylesheet	XG	Flow	Build	Analyze	Total
poem.xsl	95	0.22	0.07	0.05	0.93
AffordableSupplies.xsl	22	0.05	0.05	0.28	1.07
agenda.xsl	10	38	0.08	0.06	0.83
news.xsl	81	0.18	0.08	0.07	0.92
CreateInvoice.xsl	100	0.25	0.11	0.86	1.77
adressebog.xsl	412	0.19	0.20	0.32	1.32
order.xsl	173	0.26	0.11	0.16	1.17
slideshow.xsl	254	0.36	0.14	0.82	2.11
psicode-links.xsl	304	0.42	0.15	0.19	1.45
ontopia2xtm.xsl	318	0.34	0.20	0.83	2.08
proc-def.xsl	344	0.37	0.19	0.80	2.10
email_list.xsl	291	0.39	0.18	0.35	1.69
tip.xsl	492	0.69	0.24	0.28	1.92
window.xsl	515	0.41	1.47	3.02	5.83
dsd2-html.xsl	72,699	6.95	15.22	56.17	79.55

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

Søren Kuula's MSc thesis (March 2006):

- full **XSLT 2.0** (and thus full XPath 2.0)
- full **XML Schema**, not just DTD
- much faster:
  - weed out potential flow edges with Dong & Bailey's technique
  - avoid automata computations as much as possible

Conclusion

- The first, 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