Static Validation of XSL Transformations

Anders Møller
Mads Østerby Olesen
Michael I. Schwartzbach

University of Aarhus

http://www.brics.dk/~amoeller/talks/xslt.pdf
Plan

- Brief summary of XSLT (1.0)
- Stylesheet mining
- Type checking XSLT stylesheets
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**
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.
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:b="http://businesscard.org" xmlns="http://www.w3.org/1999/xhtml">

<xsl:template match="b:card">
    <html>
        <head>
            <title><xsl:value-of select="b:name/text()"/></title>
        </head>
        <body bgcolor="#ffffff">
            <table border="3">
                <tr><td>
                    <xsl:apply-templates select="b:name"/>
                    <br/>
                    <xsl:apply-templates select="b:title"/>
                    <p/>
                    <tt><xsl:apply-templates select="b:email"/></tt>
                    <br/>
                    <xsl:if test="b:phone">
                        Phone: <xsl:apply-templates select="b:phone"/>
                    </xsl:if>
                </td><td>
                    <xsl:if test="b:logo">
                        <img src="{b:logo/@uri}"/>
                    </xsl:if>
                </td></tr>
            </table>
        </body>
    </html>
</xsl:template>

<xsl:template match="b:name | b:title | b:email | b:phone">
    <xsl:value-of select="text()"/>
</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
Pattern Matching

- Patterns are simple XPath 1.0 expressions evaluating to node sets
  - they are (unions of) location paths
  - only child (default), attribute (@), and descendant-or-self (//) axes are permitted

- A given node $N$ matches a given pattern $P$ iff
  $$\exists \text{context } C: N \in \text{eval}(P,C)$$
Processing Modes

- `mode` attribute on `template` and `apply-templates`

- Allows a node to be processed multiple times in different ways
Variables and Parameters

- Allow **reuse** of computations and **parameterization** of template rules and of the entire stylesheet
  - values of type *string*, *number*, *boolean*, *node-set*, or *result tree fragment*
  - static scope rules, declared globally or locally
  - purely declarative

- **Declaration:** `variable / param`
- **Use:** `$x$
- **Actual parameter:** `with-param`
Given

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

assuming that $X$ is valid relative to $D_{in}$,

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

number of stylesheets

lines of code
### Complexity of Select Expressions

<table>
<thead>
<tr>
<th>Category</th>
<th>Number</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>3,415</td>
<td>31.2%</td>
</tr>
<tr>
<td>a</td>
<td>3,335</td>
<td>30.4%</td>
</tr>
<tr>
<td>a/b/c</td>
<td>1,153</td>
<td>10.5%</td>
</tr>
<tr>
<td>=</td>
<td>740</td>
<td>6.8%</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>text()</td>
<td>235</td>
<td>2.1%</td>
</tr>
<tr>
<td>a[... ]</td>
<td>223</td>
<td>2.0%</td>
</tr>
<tr>
<td>/a/b/c</td>
<td>110</td>
<td>1.0%</td>
</tr>
<tr>
<td>a[... ]/b[... ]/c[... ]</td>
<td>82</td>
<td>0.7%</td>
</tr>
<tr>
<td>@</td>
<td>68</td>
<td>0.6%</td>
</tr>
<tr>
<td>/a[... ]/b[... ]/c[... ]</td>
<td>43</td>
<td>0.4%</td>
</tr>
<tr>
<td>..</td>
<td>32</td>
<td>0.3%</td>
</tr>
<tr>
<td>/</td>
<td>8</td>
<td>0.1%</td>
</tr>
<tr>
<td>$x</td>
<td>365</td>
<td>3.3%</td>
</tr>
<tr>
<td>name known</td>
<td>250</td>
<td>2.3%</td>
</tr>
<tr>
<td>parent and name known</td>
<td>190</td>
<td>1.7%</td>
</tr>
<tr>
<td>set of names known</td>
<td>69</td>
<td>0.6%</td>
</tr>
<tr>
<td>sibling axis</td>
<td>31</td>
<td>0.3%</td>
</tr>
<tr>
<td>set of parents known</td>
<td>11</td>
<td>0.1%</td>
</tr>
<tr>
<td>parent known</td>
<td>9</td>
<td>0.1%</td>
</tr>
<tr>
<td>nasty</td>
<td>120</td>
<td>1.1%</td>
</tr>
<tr>
<td>Total</td>
<td>10,962</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
### Complexity of match Expressions

<table>
<thead>
<tr>
<th>Category</th>
<th>Number</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4,710</td>
<td>53.9%</td>
</tr>
<tr>
<td>absent</td>
<td>1,369</td>
<td>15.7%</td>
</tr>
<tr>
<td>a/b</td>
<td>523</td>
<td>6.0%</td>
</tr>
<tr>
<td>a[@...']</td>
<td>467</td>
<td>5.3%</td>
</tr>
<tr>
<td>a/b/c</td>
<td>423</td>
<td>4.8%</td>
</tr>
<tr>
<td>/</td>
<td>256</td>
<td>2.9%</td>
</tr>
<tr>
<td>*</td>
<td>217</td>
<td>2.5%</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>text()</td>
<td>52</td>
<td>0.6%</td>
</tr>
<tr>
<td>@</td>
<td>24</td>
<td>0.3%</td>
</tr>
<tr>
<td>@*</td>
<td>16</td>
<td>0.2%</td>
</tr>
<tr>
<td>a:*</td>
<td>12</td>
<td>0.1%</td>
</tr>
<tr>
<td>processing-instruction()</td>
<td>11</td>
<td>0.1%</td>
</tr>
<tr>
<td>@:</td>
<td>4</td>
<td>0.0%</td>
</tr>
<tr>
<td>a[...]</td>
<td>225</td>
<td>2.6%</td>
</tr>
<tr>
<td>.../a[...]</td>
<td>225</td>
<td>2.6%</td>
</tr>
<tr>
<td>.../a</td>
<td>108</td>
<td>1.2%</td>
</tr>
<tr>
<td>.../@</td>
<td>24</td>
<td>2.7%</td>
</tr>
<tr>
<td>.../text()</td>
<td>11</td>
<td>0.1%</td>
</tr>
<tr>
<td>.../@a:*</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>nasty</td>
<td>97</td>
<td>1.1%</td>
</tr>
<tr>
<td>Total</td>
<td>8,739</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
The XSLT Validation Algorithm

Our strategy:

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

Not supported:

- `text` output method,
- `disable-output-escaping` implementation-specific extensions
- namespace nodes with `for-each` and variables/parameters
22 steps – some highlights:

- 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
Validity Preserving Simplifications

- remove all `processing-instruction` and `comment` instructions

(we can’t ignore text since DTD can constrain attribute values)
Approximating Simplifications

12 steps – some highlights:

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

(Note: we want to handle almost-identity transformations precisely!)
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, or `value-of` 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)`
Flow Analysis

- For each `apply-templates` instruction, what are the possible target template rules?
- Which templates may be instantiated when the document root is processed?
- For each template rule, what are the possible types and names of context nodes when the template is instantiated?

- Goal: conservative approximations ("too large" is OK)
- Algorithm sketch:
  - find entry nodes (easy)
  - for each `apply-templates`, find outgoing edges and context sets (this is the difficult part!)
  - iterate until fixed point
Select-Match Compatibility

- **A necessary compatibility condition:**
  There exists an XML document $X$ valid relative to $D_{in}$ with nodes $a$, $b$, $c$, $d$ such that
  
  \[
  a \xrightarrow{\text{match}_n} b, \quad b \xrightarrow{\text{select}^i_n} c, \quad d \xrightarrow{\text{match}_m} c
  \]

  and $b$ is a node of type $\sigma$

- If this condition is satisfied for $\sigma \in \text{context}(n)$, then add an edge from the $i$’th apply-templates instruction in template rule $n$ to template rule $m$
A necessary compatibility condition (simplified): There exists an XML document $X$ valid relative to $D_{in}$ with nodes $a$, $c$, $d$ such that

$$a \xrightarrow{\text{match}_n / \text{type}(\sigma) / \text{select}_n^i} c, \quad d \xrightarrow{\text{match}_m} c$$

(assumes that select$_n^i$ does not start with ‘/’).
Decidability

- Everything has been reduced to regular tree languages and regular expressions on trees, so select-match compatibility is decidable.

- However, building an algorithm on this presumably wouldn’t be efficient...
More than 90% of all select expressions are “downwards only”!!

The set of valid downwards paths relative to $D_{in}$ is a regular (string) language – and making a DFA is easy!

Downwards XPath locations paths can be encoded as simple regular expressions!

Select-Match compatibility test:

$$\text{REGEXP}(\text{match}_n / \text{type}(\sigma) / \text{select}^i_n) \cap \text{REGEXP}(\text{match}_m) \cap \text{DFA}(D_{in}) \neq \emptyset$$
The remaining 10%?
Approximate!

Example:

\[ s_1/s_2/.../s_i/.../s_n \]

where \( s_i \) is the right-most step with a non-downwards axis

is rewritten to

\[ //s_{i+1}/.../s_n \]
Another Pragmatic Approach

- Simulate `select/match` expressions on $D_{in}$
- Add edge if non-empty intersection of results

Compared to the first approach,
- more precise on non-downwards axes
  - example:
    ```
    select = ../following-sibling::*
    match = b/*
    ```
- less precise on correlated downwards expressions
  - example:
    ```
    select = a/b/c
    match = d/c
    ```

- Only add flow edges if **both** approaches say so
Propagating Context Information

- Which contexts flow along the edges?

- For the first approach:
  *just check the incoming edges of the accept states of the resulting automaton*

- For the second approach:
  *just take the intersection of the sets that result from the select/match simulation*
Refinements

- **Modes**: only add edges/context if modes match

- **Priorities**: skip edge if always overridden

- **Predicates**: use primitive theorem prover to avoid impossible edges

- **Parameters**: global flow-insensitive (weak updates) – this also eliminates all `copy-of` instructions
We now have:

- **flow edges** *(apply-templates → template)*
- **initial** template rules
- **context set** for each template rule

Next:

- **construction of summary graph**
- **validation** relative to $D_{out}$
Summary Graphs

- Also used in JWIG / XACT program analyses

- **Nodes** ~ elements/attributes/gaps
  - *root* nodes ~ outermost

- **Edges** ~ potential “plug” operations
  - *template* edges ~ template plugs
  - *string* edges ~ string plugs, labeled with regular languages

- A summary graph can theoretically be **unfolded** to a (potentially infinite) set of concrete XML documents
Construction of Summary Graphs

Convert each template to a summary graph fragment, relative to a context:

- **element** → an element node with appropriate contents
- **attribute** → an attribute node
- **value-of** → a gap node and a string edge
- **choose** → a gap node with a template edge for each branch
- **apply-templates** → ???

- Elements/attributes whose name is `xslv:unknownString()` immediately result in **validity errors** being reported

- Initial template rules become root nodes
Connecting Summary Graph Fragments

Converting `apply-templates`:

- we have the outgoing flow edges!

- if `select` is a *children-only* step:
  - build a summary graph fragment corresponding to the *content model* of the current context,
  - connect with flow edge targets
  - if `sort` is used, scramble order

- sequence of children-only steps: handled similarly...

- `parent::node() / self::node()`: template edge to each flow edge target

- otherwise: any order and any number of occurrences of each flow edge target
Validating Summary Graphs

- **Input:** a summary graph and $D_{out}$
- **Output:** valid?

- **Solution:** use algorithm from JWIG / XACT!
Experiments

- 17 benchmarks consisting of \((\text{stylesheet}, D_{in}, D_{out})\), all written by others

- **Real errors** detected in **10** triples! (total: 44 errors)
  - misplaced elements
  - undefined elements / attributes / attribute values
  - missing elements / attributes

- **Spurious errors** detected in 6 triples
  - 90% caused by inadequate string analysis (e.g. `NMTOKENS`)

- *Soundness ensures that no errors are missed!*

- **Efficiency**: 4 minutes on a 2,528 line stylesheet with 2,561 + 1,198 line DTDs, generates summary graph with 12,182 nodes
Conclusion

- Program analyzer for statically checking validity of output of XSLT 1.0 transformations

- Main ideas:
  - reduce to core features
  - pragmatic flow analysis
  - exploit summary graph formalism from JWIG / XACT