Practical type-safe XSLT 2.0 stylesheet authoring

Søren Kuula

6th December 2006

Contents

1 Abstract

In computer programming, strict data typing is well known to safeguard against a large class of common errors, and statically verifying compilers advance the time of awareness of even more errors. XML documents can also be formally typed, and recently it has been demonstrated that sound although not complete static validation of XSLT transforms against specified input and output type descriptions (DTDs) is feasible.

The purpose of this work is to show that static validation of XSLT can be brought to the same utility in practical XSLT authoring work as that of statically validating compilers and code-assisting integrated development environments in application programming. The work extends upon previous work on static XSLT validation, refining the method towards meeting the requirements identified in a real (as opposed to purely academic) XSLT development situation: Real-time performance, confidence-building analytic precision, and operability with the standards that are believed to become dominant in the near future: W3C XML Schema and XSLT 2.0.

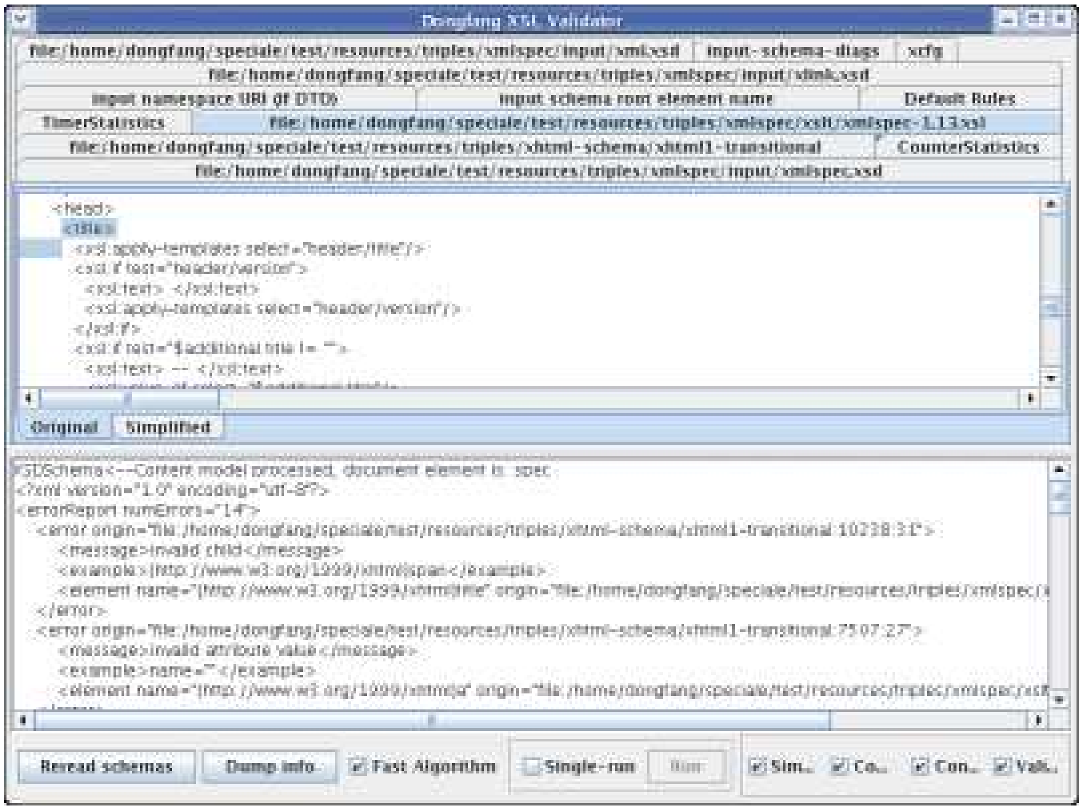
We further investigate some simpler static analysis algorithms for XSLT, and examine how they can be used for advancing the time of error awareness in practical XSLT authoring, helping to shorten development time.

Finally, we present a demonstrator application framework for experimentation with static

analysis enhanced XSLT authoring tools, and we evaluate the algorithms presented using a set of real stylesheets.

2 Contributions

- A proof-of-concept demonstration that XSLT 2.0 (XSLT2) validation and on-line code assist tools can feasibly be integrated into a development environment, like in the well-known IDEs for several other programming languages
- An extension of the only known practical XSLT validation algorithm, to
  - Work with W3C XML Schema as the input/output document type definition language, including:
    - Extending the basic algorithm from working with local-type regular tree languages to single-type (as opposed to simply using a local-type approximation)
    - Finely modeling the XML Schema data types, providing precise validation also for data
  - Work with the nontrivial new features in XSLT2
  - Perform fast enough for on-line use in an editor; much faster than the previous algorithm, despite an increase in overall complexity
  - Provably retain at least the precision of the known algorithm
- A solid, extensive (about 40k lines) quality software framework for XSLT2 simplification, analysis and validation
- A technique for statically approximating the result of template recursion on variables in XSLT2
- A single-pass, fast XSLT2 stylesheet simplification technique
- A dead code and empty-selection detection algorithm for XSLT2
- An algorithm for detecting templates in XSLT2 that never output data, neither directly or via other templates
- An algorithm detecting unused input schema declarations, for a slightly restricted variant of XSLT2. This can serve as a basis for a filter generator, helping reduce input tree size.


Dongfang XSL Validator
file:/home/dongfang/speciale/test/resources/triples/xmlspec/input/xml.xsd
input-schema-diags
xcfg
file:/home/dongfang/speciale/test/resources/triples/xmlspec/input/xlink.xsd
input namespace URI (if DTD)
input schema root element name
Default Rules
TimerStatistics
file:/home/dongfang/speciale/test/resources/triples/xmlspec/xslt/xmlspec-1.13.xsl
file:/home/dongfang/speciale/test/resources/triples/xhtml-schema/xhtml1-transitional
CounterStatistics
file:/home/dongfang/speciale/test/resources/triples/xmlspec/input/xmlspec.xsd
Original
Simplified
Reread schemas
Dump info
Fast Algorithm
Single-run
Run
Sim...
Co...
Con...
Vali...

```
    <head>
      <title>
        <xsl:apply-templates select="header/title"/>
        <xsl:if test="header/version">
          <xsl:text> </xsl:text>
          <xsl:apply-templates select="header/version"/>
        </xsl:if>
        <xsl:if test="$additional.title != ''">
          <xsl:text> -- </xsl:text>
```

```
XSDSchema<--Content model processed, document element is: spec
<?xml version="1.0" encoding="utf-8"?>
<errorReport numErrors="14">
  <error origin="file:/home/dongfang/speciale/test/resources/triples/xhtml-schema/xhtml1-transitional:10238:31">
    <message>invalid child</message>
    <example>{http://www.w3.org/1999/xhtml}span</example>
    <element name="{http://www.w3.org/1999/xhtml}title" origin="file:/home/dongfang/speciale/test/resources/triples/xmlspec/
  </error>
  <error origin="file:/home/dongfang/speciale/test/resources/triples/xhtml-schema/xhtml1-transitional:7507:27">
    <message>invalid attribute value</message>
    <example>name=""</example>
    <element name="{http://www.w3.org/1999/xhtml}a" origin="file:/home/dongfang/speciale/test/resources/triples/xmlspec/xsl
```

Figure 1: The exploratory application has found and highlighted a validation error in a stylesheet/schema combination. In the version shown, errors are presented in raw XML back-end format. About 50.000 lines of code went into the work, of which about 5000 are now legacy code, and 5000 were one-time experiments. The code is separable as stylesheet simplifier, XPath2 parser and expression model, flow analyzer, validation translator and UI.
A command line front end, and even a Web-based validator exist; the latter at `http://dongfang.dk/xslv/xslv.html`

Part I

The Scene

3 The XSLT scene

XML is probably only challenged by HTML, ASCII and English in being *the* best known language on the Web, and as its specification's 10 year anniversary closes in, few computer application domains remain with if not several competing, then least one XML language to contain their data and exchanges: XML has made its way into everything from legal court filings to hog farming, and there seems to be nothing end to that trend any time soon.

Some XML applications have evolved into entities of their own: News agencies broadcast in RSS; librarians categorize in MARC, typesetters typeset in FO and web services talk SOAP. A *name* and a *schema* defining an XML application's language, and some organization designing and managing them make them the lingua franca of the Web. Besides these standardization applications, XML is easy to use for almost any data representation: Opposite SGML, its predecessor, an XML application does not even need to be specified by a schema, and being *self-describing*, XML serves as the syntactic container of countless structured languages, replacing small languages of the old school that each had its own structure — if defined at all — down to the lexical level.

An example of an XML document, holding structured data on a company's salespeople and customers is in Figure 2.

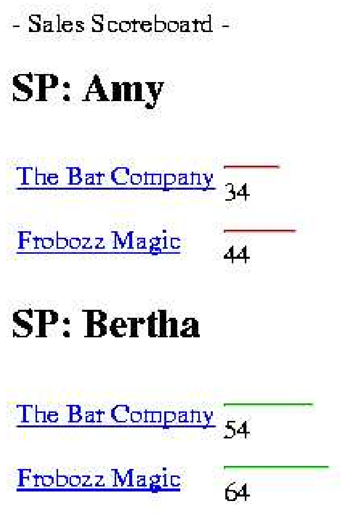
The XML language is merely syntax; any structure fitting a tree with labeled nodes can be described in XML. As [27] put it, "... these XML-related standards seem quite poor: The basic layer of the specifications just regulates the encoding of arbitrarily formed trees". There must be some way of telling what makes up an XML application, and how to decide whether a document is of the right kind or not. *Schemas* are that, as formal descriptions that are both human- and machine readable, specifying a subset of the XML-encoded trees that belong to a particular XML application. *Schema validation* is the process of determining automatically whether an XML document is *valid with respect to* a schema; if it is in the language specified by the schema. A schema for our sales report example application appears in Figure 4.

Structured data often needs to be converted to other structured data: *Data querying* is a part of that, so are the mechanics that enable *separation of data representation and presentation* and a large part of data interchanges on the Web. With structured data represented in XML, it becomes immediate to skip the distinction between the *structured data* proper with versus its *XML representation*, and just regard the task an XML to XML transformation: A language for this purpose, XSLT, was developed with Web data presentation layering in mind, and has been a World Wide Web Consortium (W3C) Recommendation since 1999. Today, its Version 1 is the most popular XML to XML transformation language, and Version 2 (XSLT2) is at the Candidate Recommendation stage, soon to replace Version 1.

```
<sales xmlns="http://www.example.com">
<spersons>
 <sperson teamcolor="ff1010">
  <name>Amy</name>
  <email>mailto:am@exa.com</email>
  <sales>
   <customer>barco</customer>
   <amount>34</amount>
   <customer>bozinc</customer>
   <amount>44</amount>
  </sales>
 </sperson>
 <sperson teamcolor="10ff10">
  <name>Bertha</name>
  <email>mailto:be@exa.com</email>
  <sales>
   <customer>barco</customer>
   <amount>54</amount>
   <customer>bozinc</customer>
   <amount>64</amount>
  </sales>
 </sperson>
</spersons>

<customers>
 <customer id="barco">
  <name>The Bar Company</name>
  <homepage>www.bar.com</homepage>
  <contact>
   <name>Bar the Barbarian</name>
   <email>mailto:bar@bar.com</email>
  </contact>
 </customer>
 <customer id="bozinc">
 <name>Frobozz Magic</name>
  <homepage>
http://www.frobozz.zork</homepage>
 </customer>
</customers>
</sales>
```

Figure 2: An example XML document: A file of salespersons and customers. Amy and Bertha both have made sales to both customers on file; the customer for each sale precedes the amount. Details of the customer sold to is in a `customer` element inside the `customers` element, identified by the `id`. The name `sales` is used in two different meanings here; one being the complete sales report, the other a list of each salesperson's sales. The `customer` name is also used both for holding details of a customer and for referring to the holder from elsewhere.



- Sales Scoreboard -

SP: Amy

The Bar Company 34

Frobozz Magic 44

SP: Bertha

The Bar Company 54

Frobozz Magic 64

Figure 3: An XHTML rendering of the sales report, as done by the transform in Figure 5.

```
<schema
xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.com"
xmlns:foo="http://www.example.com"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
<element name="sales">
 <complexType>
  <sequence>
   <element name="spersons">
    <complexType>
     <sequence>
      <element name="sperson"
      type="foo:SPerson"
        minOccurs="1"/>
     </sequence>
    </complexType>
   </element>
   <element name="customers">
    <complexType>
     <sequence>
      <element ref="foo:customer"
        minOccurs="0"/>
     </sequence>
    </complexType>
   </element> </sequence>
 </complexType> </element>

<element name="customer">
 <complexType>
  <all>
   <element name="name"
     type="string"/>
   <element name="homepage"
     type="anyURI"/>
   <element name="contact"
     type="foo:Person"/>
  </all>
  <attribute name="id" use=
    "required" type="string"/>
 </complexType>
</element>
<complexType name="Person">
 <sequence>
  <element name="name" type="string"/>
  <element name="email" type="anyURI"/>
 </sequence>
</complexType>

<simpleType name="TeamColor">
 <restriction base="string">
  <enumeration value="ff1010"/>
  <enumeration value="10ff10"/>
  <enumeration value="1010ff"/>
 </restriction>
</simpleType>

<complexType name="SPerson">
 <complexContent>
  <extension base="foo:Person">
   <sequence>
    <element name="sales">
     <complexType>
      <sequence minOccurs="1"
        maxOccurs="unbounded">
       <element name="customer"
         type="string"/>
       <element name="amount"
         type="integer"/>
      </sequence>
     </complexType>
    </element>
   </sequence>
   <attribute name="teamcolor"
use="required" type="foo:TeamColor"/>
  </extension>
 </complexContent>
</complexType>
</schema>
```

Figure 4: Example sales report schema, written in XML Schema. Elements named `sales` and `customer` are declared twice each, reflecting the different uses of them in instance documents. Team colors could arguably have been defined more readably, and a referential constraint enforcing that a customer with the right `id` exists for each `sales` would have improved the design, but it has been omitted.

```
<xsl:stylesheet version="1.0"
xmlns:xsl=
"http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:ex="http://www.example.com">

<!-- r1: document element -->
<xsl:template match="/ex:sales">
 <html>
  <head><title>Sales Report</title>
    </head>
  <body>
   <xsl:apply-templates
     select="ex:spersons"/>
  </body> </html>
</xsl:template>

<!-- r2: the sperson matches -->
<xsl:template match="ex:spersons">
 - Sales Scoreboard -
 <xsl:apply-templates/>
</xsl:template>

<!-- r3: Color table in team color -->
<xsl:template match="@teamcolor">
 <xsl:choose>
<!-- red color too ugly; change it -->
  <xsl:when test=". = 'ff1010'">
    #ff1212</xsl:when>
  <xsl:otherwise>
   <xsl:value-of
     select="concat('#',.)"/>
  </xsl:otherwise> </xsl:choose>
</xsl:template>

<!-- r4: Individual salespersons -->
<xsl:template match="ex:sperson">
 <xsl:apply-templates select="ex:name"/>
 <table>
  <xsl:attribute name="bgcolor">
   <xsl:apply-templates
     select="@teamcolor"/>
  </xsl:attribute>
  <xsl:apply-templates select="ex:sales"/>
 </table>
 </xsl:template>
```

```
<!-- r5: Name of salesperson -->
<xsl:template
  match="ex:sperson/ex:name">
 <h2>SP:
   <xsl:value-of select="text()"/></h2>
</xsl:template>

<!-- r6: Other names -->
<xsl:template match="ex:name">
 <a>
<!-- use contact mail if available -->
  <xsl:choose>
   <xsl:when test="../ex:contact">
    <xsl:apply-templates
      select="../ex:contact"/>
    <xsl:value-of select="../ex:name"/>
   </xsl:when>
   <xsl:otherwise>
<!-- otherwise use the homepage -->
    <xsl:attribute name="href">
     <xsl:value-of
       select="../ex:homepage"/>
    </xsl:attribute>
    <xsl:value-of select="."/>
   </xsl:otherwise>
  </xsl:choose>
 </a>
</xsl:template>

<!-- r7: contacts' email -->
<xsl:template match="ex:contact">
 <xsl:attribute name="href">
  <xsl:value-of select="ex:email"/>
 </xsl:attribute>
</xsl:template>

<!-- r8: sales(2) elements -->
<xsl:template match="ex:sales">
 <xsl:for-each select="ex:customer">
  <tr>
   <td>
    <!-- customers in sales elements -->
    <xsl:apply-templates select="."/>
   </td>
   <xsl:apply-templates select=
     "following-sibling::ex:amount[1]"/>
  </tr>
 </xsl:for-each>
</xsl:template>
```

```
<!-- r9: match customer in sales -->
<xsl:template match="ex:sales/ex:customer">
 <xsl:variable name="id" select="."/>
 <xsl:apply-templates select=
  "//ex:customers/ex:customer[@id=\$id]/ex:name"/>
</xsl:template>

<!-- r10: render the amount as a proportional length horz rule -->
<xsl:template match="ex:amount">
 <td><hr align="left" width="{text()}"/><xsl:value-of select="."/></td>
</xsl:template>
</xsl:stylesheet>
```

Figure 5: Example XSLT transform producing the output in Figure 6 from the XML in Figure 2. Basically, the input document is processed by the XSLT transformer first finding a `template` matching its root (r_1). The template is then *invoked* with the root node as its *context item*, causing non-XSLT elements to be copied to a result tree, and XSLT instructions (the XSLT elements inside templates) to be evaluated. Any `apply-templates` instruction selects a sequence of nodes relative to the context item, then matches each against the other templates and recurses to the one offering the best match, this time with a selected node as the context item.
This structural traversal of the input tree and outputting of nodes to a result tree goes on until all selected nodes have been processed.

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ex="http://www.example.com">
<head> <title>Sales Report</title> </head>
 <body> - Sales Scoreboard -
  <h2>SP: Amy</h2>
  <table bgcolor="#ff1010">
    <tr> <td><a href="mailto:bar@bar.com">The Bar Company</a></td>
         <td><hr align="left" width="34"/>34</td> </tr>
    <tr> <td><a href="http://www.frobozz.com.zork">Frobozz Magic</a></td>
         <td><hr align="left" width="44"/>44</td> </tr> </table>
  <h2>SP: Bertha</h2>
  <table bgcolor="#10ff10">
    <tr> <td><a href="mailto:bar@bar.com">The Bar Company</a></td>
         <td><hr align="left" width="54"/>54</td> </tr>
    <tr> <td><a href="http://www.frobozz.com.zork">Frobozz Magic</a></td>
         <td><hr align="left" width="64"/>64</td> </tr> </table>
 </body>
</html>
```

Figure 6: XHTML output of the transform in Figure 5, run on the XML in Figure 2 and whitespace-edited. This rendered to Figure 3 in a browser.

Suppose somebody wants a translation of the sales report to HTML, making the sales report readable in a Web browser. The example XSLT transform in Figure 5 does that. The transform it not a particularly pretty piece of code (but please accept for the sake of the argument that it ended up that way); one may wonder if it works at all? Whether it always works? Or whether it really produces good HTML (actually, XML HTML; XHTML) code?

XSLT1, itself an XML application, has long proved its worth as an universal expert tool in XML transformation, but it has several limitations:

- The language is inherently difficult to understand for most non-expert developers, although it seems that its expressiveness was targeted "just right"; not ridiculously strong for the purposes for which it is used, and usually strong enough not to be given up in complex cases.
- Type safety and type checkers can safeguard against a large class of programming errors: Those occurring when a data value of some kind becomes input of a computation that does not work with that kind of data. *Strongly typed languages* use *type systems* and *static analysis* to make a proof *prior to run-time* that this situation will never happen; if the proof cannot be made, the program will be rejected for execution, with an error message telling where in the program a problem was found.

  On the XML/XSLT scene, XML is the data, schemas define types and XSLT transforms describe programs, but in the present situation, although schemas may be at hand, XSLT developers do not have a practical static type checker to guarantee that their transformation output will conform to a given schema.
- Integrated development environments that support an enhanced view of code as more than just text exist for several programming languages. Several boast on-line syntactic and semantic verifiers that help programmers correct errors earlier in the process. XSLT currently lags behind on the tooling scene:
  - Most XML editors will do basic XML syntax checks and maybe schema validation, and some of the more advanced development environments for XML and XSLT provide an abstracted view of the transform, with a debugger and some analysis tools added.
  - However, the full power of on-line code validity checking and type checking algorithms that have proven extremely helpful to users of IDEs for other programming languages, such as IntilliJ, eclipse and Microsoft Visual Studio[1] is currently lacking for XSLT.

4 XSLT authoring problems

We first examine which problems seem to be the most prevalent in practical XSLT authoring, and what has been done to solve them.

[1]Bordering on forming habits: Some people get dependent on features like automatic highlighting of errors caused by edits elsewhere!

4.1 The static XSLT validity problem

The static XSLT validity problem is: Given an XSLT transformation T and the XML schemas D_{in}, D_{out}, does it hold that the transformation, when run on *any* document which is valid with respect to D_{in}, will produce a result that is valid with respect to D_{out}?[2]

The solution of this problem is interesting for several reasons:

Safety: Many programs and services accepting input in the form of XML documents have a well-defined behavior for documents that are valid with respect to a particular schema, but not for documents that are not. If using XSLT to generate such documents, it is desirable to know whether *all* of the documents are schema-valid.

Performance: If a static validity guarantee exists, then any schema validation for ensuring XSLT output validity can be removed, and with that the need to consider what should be done if schema validation fails. Programs that use output of the XSLT need not implement input validation, or have any error handling for schema-invalid input.

The static XSLT validation problem has been the subject of a great deal of research: It is nowhere near easy. The full XSLT language is Turing complete[3], meaning (Rice's Theorem) that it is impossible to soundly and completely prove any interesting property, such as validity, about XSLT stylesheets.

Researchers have taken two different approaches around this problem: Either restricting to non-Turing-complete sublanguages of XSLT, and proving decidability for those, or sacrificing decidability and make do with estimates.

Exact solutions for toy sublanguages

Milo et al.[19] were among the first to formulate exact validation, which they called type-checking, for a fragment of XSLT. Their analysis is based upon modeling the transform as a *k-pebble transducer*, and the input and output schema as the regular tree languages that we will also use. They then argue that the straightforward solution, namely finding the image of the transform represented by the transformer is not possible, because generally, the transform image is not a regular tree language and can thus not be checked for inclusion in the output regular tree language. Instead, they invert the transform and map the *output* language to a regular tree language. If that language is then a superlanguage of the input schema's language — inclusion for regular tree languages is decidable — then the image of the input language is sure to be contained in the output language.

Unfortunately, their approach is not very practical, applying only to a subset of XSLT that is so small that any such stylesheet might as well be inspected by hand. The time complexity of

[2]This extends, without any challenge, to multiple input and output schemas.

[3][16]. There even exists a universal Turing machine implemented in XSLT, at `http://www.unidex.com/turing/utm.htm`

the algorithm is hyperexponential, making validity only theoretically decidable. Tozawa[25] improved the time complexity of this to exponential, but the object XSLT sublanguage remained small.

Approximate solutions

[11] provide the ground work for some early static analysis on XSLT: Not validation, but analysis of other interesting properties of XSLT stylesheets under an assumption that XML input is valid wrt. a given schema. They used an approach of modeling XSLT stylesheets as graphs, and evaluating expressions that affect control flow *abstractly*, not over XML nodes but over their schema declarations. As a result, they were able to determine roughly which parts of a stylesheet can pass control to which parts, and, particularly interesting, which control flows seem to exist by a look at the stylesheet alone, but are always *absent* when processing XML documents that are valid with respect to a particular schema.

To a large extent, an approximative solution for the static XSLT validation problem has been found, by [23] and [20] of the University of Aarhus.

Their approach relies on a *fixed point flow analysis* on the XSLT code, given an input schema's constraints, followed by the construction of a *summary graph*, generating a superlanguage of the language that the transform may output. The summary graph is finally validated against an output schema using an algorithm from a different project, resulting in either a message that the output is always contained in the language of the output schema, or a list of error messages. The analysis is *incomplete*, meaning that the result is an estimate and is sometimes wrong, but also *sound*, meaning that the approximation is always to the side of reporting too many errors: A stylesheet that may sometimes produce invalid output is never passed as one that will not, but the converse, *spurious errors*, sometimes happens.

The algorithm is targeted at practical XSLT, and it was demonstrated to work with a set of real-world XSLT stylesheets and schemas, with a reasonable number of spurious errors. However, it is limited to working only with Version 1 of XSLT, and only with XML's rather primitive built-in schema language, DTD. It also makes rather rough approximations at places.

We will use this algorithm is the basis of our work, extending and improving it in several aspects to work in an on-line editing, next-generation XSLT context, and adding new sub-algorithms and data extractors to it.

4.2 XSLT usability problems

Two mini-surveys were conducted, to get an idea of which aspects in which the most practical problems with XSL development are encountered and how these problems can be met.

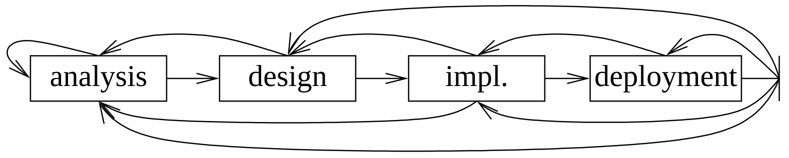


Figure 7: The sorry reality of software development: Development of any nontrivial software is a trip down a windy and bumpy road. Situations are believed to be understood, design decisions are made, and code is implemented. Often, setbacks occur: Situations are found to be different than at first thought, and things need to be re-analyzed and redesigned. Software is rarely completed without setbacks, but the smaller and the earlier, the less expensive.

Developer Interviews

A local company, using XSLT extensively in its production, agreed to participate in interviews. The company, Stibo, is in the publishing business, and receive very large XML documents from customers; one developer task is to write transforms that convert these into an in-house XML language. The in-house XML, usually representing product catalogs, is then processed (indexing, pagination etc.) and transformed to a third XML format for delivery back to the customer.

Developers had little concern for schema validity of the result of their transforms, as they were very well acquainted with the in-house schema, and almost always got correctness right by careful consideration when the transform was written, or by simply saying that their output was right, adapting their in-house tools down the pipeline to accept it. They used specialized XML editing tools like XML Spy together with common text editors for authoring transforms in XSLT1, and Xalan-j on dedicated 4 GB servers for executing them. With very large documents, a run could take more than a day to complete.

This process of ad-hoc recoding and little or no use of schemas seems very typical for the use of XML formats and XSLT in off-the-Web settings. Delivery and return schemas, when available, always were given as "hand-downs", directly from a customer with no possibility of alterations.

With recurring experiences of transforms of very large documents[4] failing after hours of execution, either because memory had run out, or because an invalid piece of transform code was finally triggered, Stibo were very interested in any tool that could enhance time and memory performance of XSLT, helping to relieve out-of-memory failures, and shortening the time before other failures eventually happen. The developers were especially intrigued by the idea of a tool that, on the basis of an input schema and an XSLT stylesheet, could generate a pre-transform in some faster transform language, removing unused subtrees from input XML trees.

[4]Stibo's developers were not aware of STX or other high-performance streaming transformer technologies would apply to their work. It seems that they will generally not, as keys (a feature incompatible with streaming) was often used.

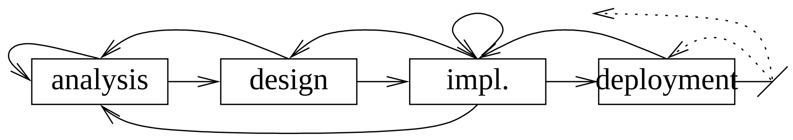


Figure 8: A wrong path can have been followed, and errors and omissions can have followed along a development process all the way from basic analysis and design; not getting aware of that until after deployment is bad quality management. Static analysis, integrated into standard development tooling, can automatically detect many erroneous situations through design and implementation, helping increase quality and saving last-minute repairs.

Internet forums

Another survey was made in the `comp.text.xml` newsgroup on Usenet: The newsgroup is a forum for general XML questions and answers. Questions related to XSLT were isolated, and divided into rough categories. The result is in Table 1.

Obscurity of control flow

For the uninitiated, and even for the advanced XSL developer, probably the hardest problem in analyzing an existing XSL transform is to comprehend the possible control flow of the transform: Which templates may pass control to which, and in which context?[5]

Common XSLT debuggers can visualize the control flow for a transform of a single instance document, and so can can the well-known "hack" of adding some text in templates for traceability. These approaches, however, require that an input instance input is available to test with, and it can be a time-consuming affair to come up with one that will trigger exactly the

[5]Anyone disagreeing is encouraged to download W3C's latest xmlspec XML Schema and to-XHTML XSLT stylesheet (or pick the files from the project programs, at test/resources/triples/xmlspec), and try to figure how it works.

OPs problem	Count
Describing a desired transformation, requesting complete solution	12
Related to XSLT whitespace handling	3
Related to XML namespaces in XSLT and XPath	2
How to select [nodes with some property] in XPath	19
How to collect all [nodes with some property] in a group	6
Meaning and context dependency of `position()` and [n]	2
Related to other XPath functions	3
Other XSLT problems	10
XSLT related, not problems	13

Table 1: Three months of XSLT-related threads on the `comp.lang.xml` Usenet newsgroup, divided into rough categories.

situation to be checked for, write it, and step through the debugger to the stage of interest. Debuggers give no clue what the template control flow may be for other documents than the specific one used in the debugging session, and complex debugging sessions can be easy to lose overview of. Code repair made after debugging also tends to be bad, often introducing "special cases" and complicating the structure of the code further. On the other hand, debuggers are valuable for inspection of value computations.

Static analysis of XSL, where the input schema is taken into account (and even where it is not) is particularly suited as a basis for a tool computing and visualizing the possible control flow between templates. Ultimately, a static analysis based tool could give the stylesheet author the debugging insight of "all instances at the same time", it could save him the trouble of writing debugging instances. Probably the most important benefit is: It can help advancing much of the awareness of the real (as opposed to what the developer had imagined) control flow in a stylesheet from *test* time to *edit* time, and awareness of errors from *deployment* time to test time, which, for a sufficiently fast analysis, could be the same as edit time.

In design rather than analysis work, we contend that a static, on-line flow visualizer built into an editor can significantly shorten the cycle time of trial and error development, by updating displayed feedback immediately, instead of the usual "edit-save-try" cycle. The first category of our `comp.text.xml` survey indicates that XSLT design is *difficult*, at least for the beginner, or maybe that XSLT authors generally are lazy. The latter would contrast with other Usenet computer programming groups, where people rarely ask for complete solutions.

Obscurity of XPath

One reoccurring problem, partially linked with the control flow problem, is the comprehension of XSLT's expression language, XPath, with an evaluation model that is little used elsewhere. [26] has written a whole research paper trying to make a sensible meaning out of the way XPath *patterns* (a restricted kind of expressions) were defined in a draft of the XSLT Recommendation, and did not neglect to mock other semanticists for publishing code libraries for the same purpose on a misunderstood foundation[6].

A more practical problem with XPath is the rather confusing way that XML namespaces are referred to from within expressions: Here, an on-line tool displaying roughly what each expression may select would be really useful.

XPath generation tools exist now in several XML IDEs, enabling point-and-click assembly of basic XPath expressions, but their expressiveness is rather limited.

The Usenet survey speaks for hardness of XPath, too: Questions about how to construct XPath expressions were the most asked of all XSLT questions.

[6]See [26], p. 10, on [Wallace and Runmican].

Understanding functional programming

XSLT is designed over a functional paradigm, with structural template recursion, and variables that are assignable only at the beginning of stylesheet or template invocation. Learning to master the tricks of the functional programming trade has to be overcome by the programmer, no matter what. Anyway, with the loose, structural matching of patterns, a flowgrapher tool of course is a help.

4.3 Existing XSLT tools

Altova's XML Spy [7] is probably the best known XML-specialized editor today. It features XSLT editing under on-line tool tip guidance, and different abstracted views on stylesheets. Schema-aware XSLT2.0 editing, debugging and transformation is now supported, but transformations apparently only work from within the IDE. It also boasts the "usual" XPath1 and XPath2 point-and-click prototypical expression generators and an XPath evaluator, but no static analysis features.

StyleVision, also from Altova, features generation of XSLT code for a fixed selection of output languages, and random XML input.

Stylus Studio[8] has about the same features as XML Spy, with a couple of interesting additions: An XSLT profiler, and a data-oriented debugger that will back-map, at a mouse click on some XSLT output, to the XSLT instruction that created it. Like XML Spy, Stylus Studio has schema-supported code views and completion and a dynamic debugger, but no static analysis, much less validation.

5 Goals and restrictions

We base our proposal for a tool improving the XSLT authoring scene simply on:

- What we believe will be useful in a practical type-safe XSLT 2.0 authoring context
- What we found to be possible, based on extending the [23] algorithm

and disregard considerations about general features that are not related to the algorithm, and about general usability: Although all this certainly is important for the success of a computer tool, it is not a central subject of this work.

[7] `http://www.altova.com/products_ide.html/`
[8] `http://www.stylusstudio.com/`

5.1 Proposal — An on-line analyzing XSLT editor

The algorithm by [23]/[20] will do basic validation of practical XSLT, and some data extraction can easily be added.

Summarizing the problems in practical development of reliable XSLT that we will try counter by building tools for them:

- Find (on-line, during edit) errors that make some transformation results become invalid wrt. to the output schema.
- Illustrate flows of control and nodes between templates (on-line, during edit).
- Illustrate apparently plausible, but nonexistent flows (on-line, during edit). This is complementary with the above.
- Detect XPath node selection expressions that never select or match anything.
- Detect dead code, and detect control flows that never reach anything contributing to output.
- Boost performance: Generate pre-transforms in a streaming transformation language, cutting away data that is never used by the main transform.

The list of other niceties to include in the ultimate XSLT tool has no end: XSLT generators generating XSLTs for a set of prototype input and output documents (there are infinitely many solutions for each case), XSLT compilers that compile faster XSLT processors for one input language and one transform — or a compiler compiler, generating processors for one input language, any transform. Output schema inference engines for input schemas and transforms, or the other way around, also would be nice to have; so would a transform class analyzer, that could translate part of all of a transform to a simpler and faster transformation language. All this falls beyond the intended scope of this project.

Requirements for a practical tool

There are some basic requirements for a tool to be appreciated and used at all:

1. It must be applicable for real-world tasks. For our XSLT tool, that means that the tool:
   - Must work with the XSLT Version 2.0 language
   - Must work with the most widely used schema languages
2. It must appear to be of more help than trouble:
   - It must be able to actually mitigate at least some of the user's doubts and concerns.

    - It must perform well, without annoying sluggishness.
    - It must deliver easily understandable results.
    - It must not force the user to think of everything in a new and different way[9].

3. It must be able to compete with its alternatives. For the XSLT setting, they are:
    - Debuggers
    - Manual trial runs

The overall problem of this work is to show that it is feasible to construct a practical tool, integrated into an editor, that fulfills all this.

Although XML DTD[5] is surprisingly capable of describing most of the schemas used[10], is can be predicted to become gradually phased out[11]. A large number of proposals for stronger XML schema languages have emerged. Of these, W3C's XML Schema Description (XSD: [1][2] and [3]) and OASIS' Relax NG[4] have become the most widely used, with XML Schema as the officially endorsed W3C Recommendation.

Because the XSLT language is now facing a transition from Version 1 to Version 2, we believe that the claim will only hold if XSLT2 is supported. Also, since the most widely schema language in the foreseeable future seems to be the World Wide Web Consortium's (W3C) XML Schema[12], and since XSLT2 itself to some extent depends on XML Schema, we will extend the [23] static XSLT validation algorithm to work with that schema language.

5.2 Goals

The core claim is that the [23] static validation algorithm can be extended to:

- Work with XSLT 2.0
- Work with XML Schema, without approximating the type systems for structures, and with only very few approximations of data types

[9]Many problems have a brilliant solution that is based upon redefining the problem entirely, but that approach is not very marketable: People are not generally not willing to relearn.

[10]In a survey by [10], 85% of all valid XML Schema's examined were of a language class that could as well have been expressed in DTD, leading to the question whether the better expressiveness of XML Schema is generally not needed, not understood, or both. We hope that the answer is "not yet understood".

[11]Even if XML application designers should for the most part not need a very strong language class, they will be needing namespace support for interoperability, and many applications will undoubtedly be designed to work with some of the XSD-specified mainstream data format application like that of Microsoft Office.

[12]It is not easy say anything definitive about the future, and the future schema situation in particular. Certainly, most XML developers see DTD as being too limited, and its weakness of not supporting namespaces as severe. XML Schema is the official Recommendation from the organization that defines XML, but it is under fire from many critics for being too complicated compared to its expressiveness, and for not being completely well-defined. However, in practical XML use, it appears to be very popular, and almost all business and industry standards that include a XML schema use W3C XML Schema.

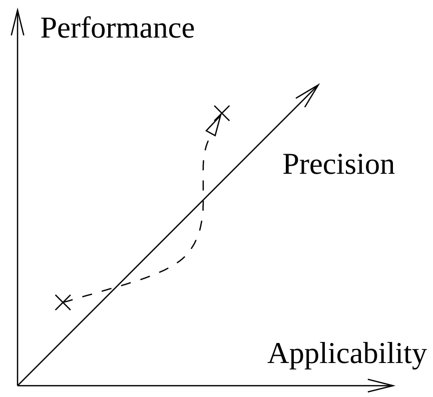


Figure 9: We improve the existing XSLT validation algorithm in three dimensions: Its application area (to XSLT2 and XML Schema), its precision and its performance.

- Perform acceptably in time and memory, even for large stylesheets and large schemas.
- Apart from validity, also supply other valuable, nontrivial input schema-sensitive code-assist information.

We will seek to show this, and:

- Develop a exploratory XSLT editor application, demonstrating that the technique is practical.
- Perform, analyze and evaluate test runs on real stylesheets.
- For features which are not in common use yet, construct test cases.

5.3 Non-goals

Explicit non-goals are:

- A formal proof of the correctness of our approach
- A complete tutorial in all the technologies involved: The reader is assumed to know basic XML, XML DTD and XSLT1. XML Schema, XSLT Version 2 changes and XPath Version 2 are introduced briefly, though.
- A complete production-quality XSLT editor implementation.
- Detailed consideration of every aspect of every feature of the languages involved.

5.4 Structure

The rest of the work proceeds as follows: First, the most important technologies on the XML and XSLT scene are briefly introduced, with some emphasis on XML Schema, since this Recommendation is so hard to comprehend that we will try to capture its meaning in our own framework, avoiding any reference to the formulations in the Recommendation. XML's own schema language, DTD, has also been included in the framework (and in the exploratory application program), with a conceptual model that covers both. The reader is assumed to know basic XML and XML Namespaces.

The largest section is dedicated to *flow analysis*, because a redesign of this was one of the most important contributions of this work, and it was here some interesting new discoveries were made: We have improved the best known flow algorithm significantly in all major aspects.

A section was devoted to following up on the different *applications* of the result of flow analysis — first and foremost validation, and also all the smaller algorithms for editor enhancements.

Finally, we present evaluation data and a conclusion.

Part II

XML Technologies

We start out by introducing the XML technologies and model used, and then present a unified model of the two most widely used schema languages, DTD and XML Schema, providing the necessary schema information for the central XSLT flow analysis algorithm presented next.

6 XML schema languages

[W3C XML Schema] is one spec where your eyes will still twitch and your head still go buzzZZZ even when you read it for the tenth time. Except that you will no longer be surprised by surprises.

Dr. Michael Kay

Since XML is just a markup syntax for ordered, labeled and unranked trees with character data, the different schema languages have one thing in common: They are representations of some class of tree languages. Schema languages feature constraints on several aspects, some going beyond tree languages:

- Structure of *markup* — determines the (tree) structure of instance document markup

- Sublanguages for *data* — determine the allowable data types for text and attribute leaves in instance documents
- Constraints on *integrity* — specify referential or uniqueness constraints in instance documents
- *Other* constraints, often called “business rules”

We will focus on the markup structure and data language aspects here, and ignore integrity and other constraints entirely: Firstly, the former apply universally in XML, whereas the latter are rarely used, and secondly, the latter are notoriously hard to analyze statically.

[22] provides a foundation for the definitions and properties of the *regular tree languages* subset of tree languages. [21] is an early characterization of the most common schema classes, along with closure properties and (non-static document) validation algorithms for them. The article also suggests an interesting, powerful and efficient yet unexploited class of regular tree languages called *restrained-competition languages.*

[18] gives a nice, brief analysis of the properties of some of the most popular schema languages classes, and proposes an alternative characterization of restrained-competition languages, *one-pass pre-order typeability*, based on the largest class that can be efficiently validated and typed.

We will relate the findings of the above papers to the way in which we use them, introduce a consistent terminology. Therefore, we will begin by analyzing the properties of DTD and W3C XML Schema that will be useful later on, in tree language terms. Finally, we will introduce definitions for a unified DTD and XML Schema model, also for later use.

In describing XML content in the following, we will ignore all other kinds of nodes than elements. We will make up for that when re-introducing them later. This is not hands-waving at inadequacies of the model with respect to “real” XML schema classes[13] — using a large enough hammer, all node kinds could be slammed into it — but as we shall see, it is not necessary.

6.1 Σ-TREES

We begin with a way expressing the structure of a tree, and identifying its nodes. Since we are dealing with schemas, declaring a finite number of names, let Σ be a finite alphabet of names.

Let $\mathcal{T}_\Sigma$ be the set of Σ-trees:

1. leaf: $\sigma \in \Sigma \Rightarrow \sigma \in \mathcal{T}_\Sigma$

[13]Well, almost. Because of its subtyping feature with `xsi:type`, XML Schema is only single-type if attributes *are* considered. We will restrict the discussion for XML Schema by assuming no subtype substitution, and repair later.

2. internal node: $\sigma \in \Sigma$ and $t_1, \cdots, t_n \in \mathcal{T}_\Sigma \Rightarrow \sigma(t_1, \cdots, t_n) \in \mathcal{T}_\Sigma$.

We do not need to distinguish between nodes and the subtrees they root.

Next comes the *domain* of a tree $t \in \mathcal{T}_\Sigma$ as a string of natural numbers:

$$\begin{array}{lcl} \text{Dom} & : & \mathcal{T}_\Sigma \to \mathbb{N}^* \\ \text{Dom}(t) & = & \{\epsilon\} \cup \{iu | i \in \{1, \cdots, n\}, u \in \text{Dom}(t_i)\}^{14} \end{array}$$

That is, the root node of t is identified by the string ϵ, the Dom applies recursively to each child, and each child t_i's ordinal i is prepended to all members of $\text{Dom}(t_i)$. All children of each node share a common prefix, and siblings each have a unique, 1-symbol suffix. In the following, we will not distinguish between a node and its identifier string.

Each node has a *name*; for XML elements, it is just the element name:

$$name^t : \text{Dom}(t) \to \Sigma$$

The t qualifier in the function name indicates that it varies with each tree instance.

For each node u in a tree t, we may define its *ancestor string*: The string of names (recall that, for our purpose, a name is a symbol in the alphabet Σ, not a string) encountered through a descent from the root of t to u:

$$\begin{array}{lcl} ans^t & : & \text{Dom}(t) \to \Sigma^* \\ ans^t(n) & = & \begin{cases} name^t(\epsilon) & \text{if } n = \epsilon \\ ans^t(n_0 \cdots n_{|n|-2}).name^t(n) & \text{otherwise} \end{cases} \end{array}$$

Complementary to ancestor strings, the *child string* of node v is the string of the names of its children:

$$\begin{array}{lcl} chs^t & : & \text{Dom}(t) \to \Sigma^* \\ chs^t(v) & = & name^t(v1). \cdots .name^t(vn) \end{array}$$

It should be easy to see that this is sufficient to describe the elements in XML trees.

6.2 Types and states

Restricting from *any* Σ-trees to trees in some tree language, need a way of describing the permissible relationships between nodes, their names, and other nodes and their names. A "tree automaton" perspective as by [22] and a "extended context-free grammar" perspective by [21] both have in common that a *state assignment* is performed on the nodes of an instance tree. The automaton perspective is easier to reason about formally:

Tree automaton perspective

Define an alphabet Σ' of *states*. A *nondeterministic regular tree automaton* is a tuple

$$A = (\Sigma', \Sigma, \delta, F)$$

where Σ is as before, $F \subseteq \Sigma'$ is a set of *final states*, and δ is a function from a node's (tentatively) assigned state and its name, to a regular expression over the permissible strings over its child nodes' assigned states:

$$\delta : \Sigma' \times \Sigma \to \mathbf{Reg}(\Sigma')$$

Now, if there exists a function λ, assigning states to all nodes:

$$\lambda : \mathrm{Dom}(t) \to \Sigma'$$

so that for every node v with n children $\in \mathrm{Dom}(t)$:

$$\lambda(v1) \cdots \lambda(vn) \in \delta(\lambda(v), name^t(v))$$

then the tree is *accepted* by the automaton. In some regular tree languages, there may be more than one choice of λ for the same tree.

Most XML schema languages express regular tree languages; reformulating the description slightly in the following section, it becomes easier to see.

6.3 A context-free grammar perspective

A slightly different perspective on regular tree languages uses extended context-free grammars to define them:

$$G = (\Sigma', \Sigma, S, P)$$

where Σ' and Σ are as in the automaton perspective, and are here called the *nonterminals* and the *terminals*, respectively. The set of final states is now called S, the set of *start nonterminals*, and the function δ is replaced by a set P of *productions* of the form $q \to \sigma r$, where $q \in \Sigma'$, $\sigma \in \Sigma$ and $r \in \mathbf{Reg}(\Sigma')$:

It can be assumed that there are no two productions in P with the same σ and q, like:

$$\{q \to \sigma\ r_1, q \to \sigma\ r_2\} \cup P_{more}$$

since that pair of productions without loss of generality can be rewritten to

$$\{q \to \sigma(r_1|r_2)\} \cup P_{more}$$

An *interpretation* of a tree t against the grammar G is still a state (nonterminal) assignment

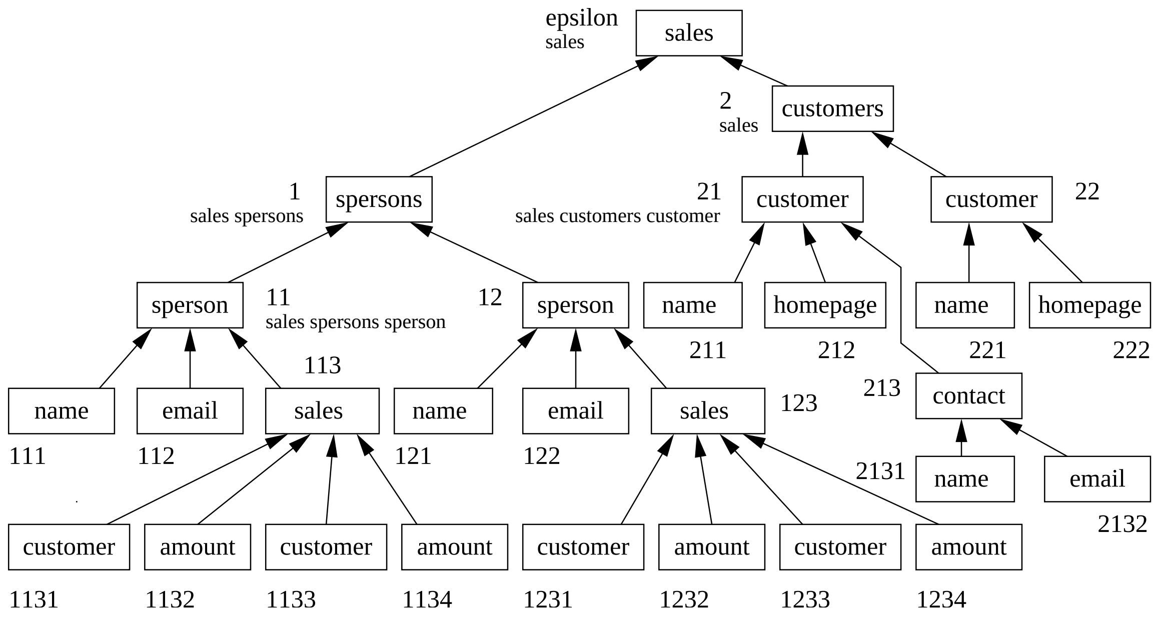


Figure 10: A Σ-tree, with the label, domain and some ancestor strings written out. The tree is the XML element tree of Figure 2.
Notice that ancestor strings do not uniquely identify nodes — for example, the `name` nodes of salespersons all have the same ancestor string. Domain labels are unique, though.
The tree is accepted by the grammar in Figure 11.

$$
\begin{aligned}
\Sigma' &= \{\mathbf{Sales}^1, \mathbf{Spersons}, \mathbf{Customers}, \mathbf{Sperson}, \mathbf{Customer}^1, \mathbf{Name},\\
&\quad \mathbf{Email}, \mathbf{Sales}^2, \mathbf{Customer}^2, \mathbf{Homepage}, \mathbf{Contact}, \mathbf{Amount}\}\\
\Sigma &= \{\texttt{sales}, \texttt{spersons}, \texttt{sperson}, \texttt{customers}, \texttt{customer},\\
&\quad \texttt{name}, \texttt{email}, \texttt{homepage}, \texttt{contact}, \texttt{amount}\}\\
S &= \{\mathbf{Sales}^1\}\\
P &= \{
\end{aligned}
$$

$$
\begin{array}{llll}
\mathbf{Sales}^1 & ::= & \texttt{sales} & \mathsf{Seq}(\mathbf{Spersons}, \mathbf{Customers}),\\
\mathbf{Spersons} & ::= & \texttt{spersons} & \mathsf{Card}(1, \mathsf{unbounded}, \mathbf{Sperson}),\\
\mathbf{Customers} & ::= & \texttt{customers} & \mathsf{Card}(0, \mathsf{unbounded}, \mathbf{Customer}^1),\\
\mathbf{Sperson} & ::= & \texttt{sperson} & \mathsf{Seq}(\mathbf{Name}, \mathbf{Email}, \mathsf{Card}(1, \mathsf{unbounded}, \mathbf{Sales}^2)),\\
\mathbf{Customer}^1 & ::= & \texttt{customer} & \mathsf{All}(\mathbf{Name}, \mathbf{Homepage}, \mathbf{Contact}),\\
\mathbf{Sales}^2 & ::= & \texttt{sales} & \mathsf{Seq}(\mathbf{Customer}^2, \mathbf{Amount}),\\
\mathbf{Contact} & ::= & \texttt{contact} & \mathsf{Seq}(\mathbf{Name}, \mathbf{Email}),\\
\mathbf{Name} & ::= & \texttt{name} & \epsilon,\\
\mathbf{Email} & ::= & \texttt{email} & \epsilon,\\
\mathbf{Homepage} & ::= & \texttt{homepage} & \epsilon,\\
\mathbf{Customer}^2 & ::= & \texttt{customer} & \epsilon,\\
\mathbf{Amount} & ::= & \texttt{amount} & \epsilon
\end{array}
$$

$$\}$$

Figure 11: A regular tree grammar, accepting the tree in Figure 10. With some effort, it can be seen to be equivalent with the XML Schema in Figure 4 , still regarding only elements, and fixing the choice of root element to `sales`.

λ to each node, such that:

$\lambda : \mathrm{Dom}(t) \rightarrow \Sigma', \lambda(root(t)) \in S$

For each $v \in \mathrm{Dom}(t)$: There exists a production rule $r \in P$:

$\lambda(v) \rightarrow lab^t(v)\ r$, such that$\lambda(v1). \cdots .\lambda(vn) \in \mathcal{L}(r)$

A tree t is *generated* by a grammar G, if there is an interpretation of t against G.

6.4 Regular tree languages vs. schema definitions

The automaton and the regular tree grammar perspectives are, of course, the same thing: Each production $q \rightarrow \sigma r$ can be translated to a δ-mapping $\delta(q, \sigma) = r$, and for all absent (q, σ) pairs, δ maps to $\emptyset$, and automaton definitions translate to grammars with the same ease. If only considering structure[15], an XML document is valid with respect to a schema if it is accepted by schema's tree automaton cf. the schema's grammar.

Both DTD and XML Schema are declarative, *declaring* elements, attributes and their content. Declarations in these schema languages translate to automaton states, or nonterminals: They apply to nodes of a certain name in a certain context, and they specify a set of acceptable content sequences for each node. We call all declarations in a schema Σ_d, and it will become evident that this is just an extension of Σ'.

XML permits most, but not all characters in the Unicode alphabet: Let **char** be the production production of the same name in [5], and **char*** be any XML string.

Element *content models* — models of the allowed child element content of elements — are regular in both DTD and XML Schema, and the regular expression language

$$\begin{array}{lcll}
\mathbf{M} & ::= & \mathsf{All} & (\mathbf{M}^+) \\
& | & \mathsf{Seq} & (\mathbf{M}^*) \\
& | & \mathsf{Choice} & (\mathbf{M}^+) \\
& | & \mathsf{Card} & (min, max, \mathbf{M}) \\
& | & q & \in \Sigma' \\
& | & \epsilon &
\end{array}$$

where $min \in \mathbb{N}_0$, and $max \in \mathbb{N}_0 \cup \{\mathsf{unbounded}\}$.

can describe the content models of both DTD and XML Schema (Card is cardinality, meaning $min \cdots max$ repetitions. Text, comment and processing-instruction content has been left out, and will be added later).

Both schema languages can have their *attribute models* adequately described with the language

[15]As opposed to also considering data values, referential constraints etc. expressed in the schema.

$$\begin{array}{llllllll} \mathbf{A} & ::= & \mathsf{Required} & (q & \in & \Sigma', S & \in & \mathbf{Reg}(\mathbf{char})) \\ & | & \mathsf{Optional} & (q & \in & \Sigma', S & \in & \mathbf{Reg}(\mathbf{char})) \\ & | & \mathsf{Fixed} & (q & \in & \Sigma', s & \in & \mathbf{char}^*) \end{array}$$

by associating a set of attribute uses, each with an A-description, with each element declaration. In DTD and XML Schema, character *data models* are regular, or approximately regular. Character data appear in attribute values, and in element content.

In XML Schema, the combined content, data and attribute models of an element declaration is called a *type*, and to avoid confusion, we will also use the word with that meaning (even for DTD).

We will use the term *declared node type* (DNT) for the nodes that are declared in declarations: A DNT is a name associated with a type. Introduce the functions:

$$\begin{array}{lll} C_m : \Sigma' & \rightarrow & \mathcal{L}(M) \\ A_d : \Sigma' & \rightarrow & 2^{\Sigma'} \\ A_m : \Sigma' & \rightarrow & \mathcal{L}(A) \end{array}$$

and

$$D_d : \Sigma' \quad \rightarrow \quad \mathbf{Reg}(\mathbf{char})$$

that return the content model for a node (for any other nodes than root and element nodes, it returns $\emptyset$), the set of declared attributes for elements ($\emptyset$ for non-elements), and a data model, respectively.

The function m simply extracts the symbols mentioned in an M-expression[16]:

$$\begin{array}{lll} m(\mathsf{All}(M_1 \cdots M_n)) & = & \bigcup_{i=1}^{n} m(M_i) \\ m(\mathsf{Seq}(M_1 \cdots M_n)) & = & \bigcup_{i=1}^{n} m(M_i) \\ m(\mathsf{Choice}(M_1 \cdots M_n)) & = & \bigcup_{i=1}^{n} m(M_i) \\ m(\mathsf{Card}(min, max, m') & = & \begin{cases} \emptyset & \text{if } max = 0 \\ m(m') & \text{otherwise} \end{cases} \\ m(q \in \Sigma') & = & \{q\} \\ m(\emptyset) & = & \emptyset \end{array}$$

the function $C_d : \Sigma' \rightarrow 2^{\Sigma'}$ returns the declared children of a DNT:

$$C_d(q) = m(C_m(q))$$

and the function $P_D : \Sigma' \rightarrow 2^{\Sigma'}$ returns the possible parent DNTs for a DNT:

$$P_d(q) = \{q' : q \in C_d(q') \mid q \in A_m(q')\}$$

[16] $\mathsf{Card}(.,0,.)$ may appear in a derivation by restriction.

6.5 DTDs as regular tree languages

A *Document Type Definition*(DTD) is (as far as elements are concerned) a list of element declarations:

`<!ELEMENT` **name content** `>`

where **content** is an one-unambiguous regular expression over element names, declaring the content with cardinality and order constraints. Alternatively, **content** is a list of element names, headed by a special token #**PCDATA**, that only restricts the allowed names of child elements (no restrictions apply on cardinality or order), and allows character data contents interspersed between child elements. There is also an **EMPTY** token, allowing no content at all, not even a comment or a processing instruction (these are otherwise allowed to appear anywhere outside tags)[17], and an **ALL** token, that works like a #**PCDATA** with a list of every *declared* element name.

It is only allowed to have one declaration for each element name in a DTD. Indeed, in DTD, for any element node in an instance document, the only thing determining the allowed content of the node is the name of the element node.

The effect of these state insensitive, *local* declarations of element contents is that the definition of the corresponding tree automaton can be restricted: Since there is exactly one regular expression over Σ' for each name in σ', we can do away with state assignment entirely, and just use names:

$$\delta_{DTD} : \Sigma' \rightarrow \mathbf{Reg}(\Sigma')$$

A validation run on a DTD-declared XML instance tree is, then, only a matter of checking, for each node v, that:

$$chs^t(v) \in \delta_{DTD}(name^t(v))$$

6.6 W3C XML Schema

An XML schema (XSD) instance consists of a number of *schema documents*, each containing a number of *schema components*. A schema component is:

- A target namespace
- A number of declarations:
    - *Element declarations* associate element names with type definitions

[17]This seems overly strict, and was probably the result of a slip in the XML specification. [23] went a long way to validate comments and processing instructions pedantically; we will be a little more relaxed on the subject, not considering comment and processing instructions output in validation.

    - *Attribute declarations* associate attribute names with type definitions
    - *Other declarations* that are not relevant in this treatment

- A number of definitions:
    - *Simple type definitions* define data model types
    - *Complex type definitions* define content model and attribute model types
    - *Model group definitions* define (parts of) content models
    - *Attribute group definitions* define sets of attribute declarations
    - *Other definitions* that are not relevant in this treatment

- A number of *include/import/redefine* references to other schema documents

- Some "helper" components (annotations, etc...)

Complex type definitions (still considering only element content) and model groups are composed of:

- *Sequence* compositors, translating to the Seq nonterminal of our M-language

- *Choice* compositors, translating to the Choice nonterminal of our M-language

- *Declarations*, possibly as references, translating to symbols in Σ'

- *Wildcards*: `any` represents any element with any name and any content. It is to some degree possible to limit the allowed namespaces of the represented elements.

- *Cardinality ranges*, two numbers `minOccurs` and `maxOccurs` may be applied to any of the above. `minOccurs` ranges from 0 to infinity, and `maxOccurs` from 0 to infinity, plus the special value unbounded. These translate to the Card nonterminal of our M-language

- *Model group references*, referring to a model group (they can be expanded syntactically)[18]

- Special *All* compositors, that represent any permutation of their contents. These are heavily restricted to appear only directly under a type definition (after model group expansion), and only contain declarations, and only with a cardinality from 0 to 1. They translate to the All nonterminal of our M-language

For some reason or another, only compositors and not *particles* like declarations and wildcards, are acceptable as the outermost component of a model group or a complex type.

The names of element declarations, the names of attribute declarations, the names of type definitions, the names of model group definitions and the names of attribute group definitions all belong in distinct *symbol spaces*, preventing name conflicts between one kind and another.

[18]There is a nice way to handle them more elegantly: Considering them nonterminals without a terminal in their production. In our implementation, they are just expanded syntactically, though.

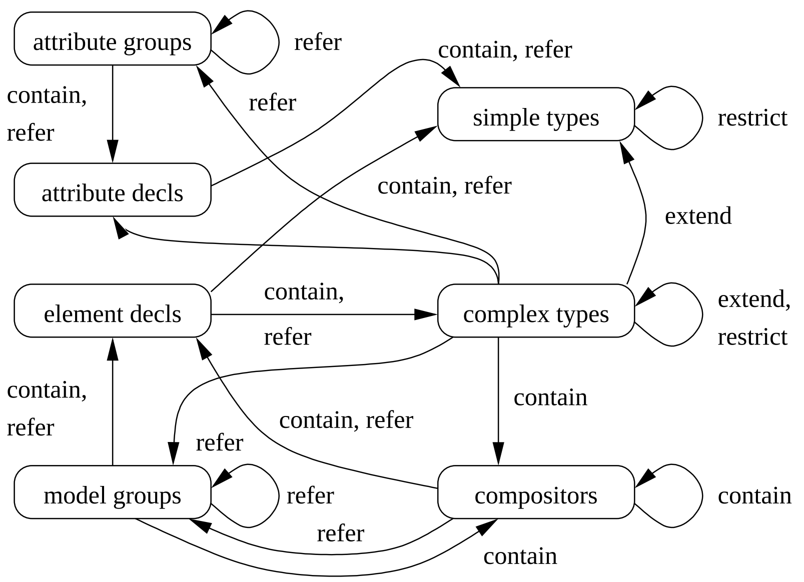


Figure 12: An attempt to sort out the confusing multitude of containment and reference structures of XML Schema. Edges describe a relationship between the schema component at the tail of the edge, and that at the head; for example, complex type definitions may extend simple type definitions. **compositors** are `sequence`, `choice` and `all`. Notice the containment cycle through **element decls**, **complex types** and **compositors**, permitting arbitrary nesting of those schema components.

Declarations and definitions may appear at different places:

- *Globally*, at *top-level* in a schema document, meaning that they are named and accessible throughout the schema. The name of a global declaration is the name of the element or attribute it declares, and there may be at most one global declaration per symbol space for any name. The name of a globally declared type has no semantic significance. The namespace of them all is always the target namespace of the schema (even for attributes).
- Element or attribute declarations may appear *locally*, *contained in complex type definitions*. Alternatively, there is a *reference* mechanism available for referring to a global declaration by its name.
- Type definitions may appear *locally*, *contained in element or attribute declarations*. Alternatively, there is another *reference* mechanism available for referring to a global definition by its name. Of course, the type definition used for an attribute declaration must be simple.
- Model and attribute group definitions are uniquely named; they may appear only at top-level, and may be referred to from within any type or group definition.

The possibility to opt between global or local declarations and definitions leaves XML Schema authors with the freedom (and burden) of deciding where to put them.There is, however, a slight but basic semantic difference between the two: Element declarations contained in type definitions can re-declare element names, even if those names are declared elsewhere (as opposed to global declarations, where any name in a symbol space may be declared at most once).

To be more specific, let us assume that model groups are dealt with for now by expanding them syntactically, that type derivation is not used, and look at the set of all (global and local) complex type definitions. The analogy to our definitions of regular tree languages now begins to take shape, if we let Σ be the names of the elements declared in a schema, and assign one symbol in Σ' to each element declaration (resolving element references) : An element declaration becomes a triple of

$$(\text{its state } q \in \Sigma',\ \text{its element name } \sigma \in \Sigma, C_d(q))$$

that can be written into a mapping of a δ function, or an extended context-free production, in the obvious way.

There is a opportunity for clarification calling at us here: We might as well use the element declarations as the set of states cf. nonterminals as the state names:

$$q \mapsto e^i_\sigma | q \text{ is the } i\text{'th declaration (in some order) of an element named } \sigma$$

and define a function μ that maps declarations to the name of the node type declared:

$$\begin{array}{lcl} \mu : \Sigma' & \rightarrow & \Sigma : \\ \mu(e^i_\sigma) & = & \sigma \quad , \quad \mu(a^i_\sigma) = @.\sigma \quad , \quad \mu(\mathbf{root}) = \mathbf{root} \\ \mu(\mathbf{pi}) & = & \mathbf{pi} \quad , \quad \mu(\mathbf{comment}) = \mathbf{comment} \quad , \quad \mu(\mathbf{pcdata}) = \mathbf{pcdata} \end{array}$$

In contexts where a variable ranging over Σ' is used, and the symbols σ or i are not bound or have any significance, we will still use just the name q.

DTD is local-type — XML Schema is not as limited: In some XML Schema instances, there are more than one declaration with the same element name. On the other hand, care has been taken by the designers of XML Schema to make elements typeable already as their opening tags are encountered by an XML parser.

The XML Schema Recommendation[2] specifies two *Schema Component Constraint*s on content models (restated here in a simpler language than in the Recommendation):

- *Element Declarations Consistent (EDC)*: If there is, within the same type definition, a pair of declarations of elements with the same name, then both declarations must use the same global type.
- *Unique Particle Attribution (UPA)*: It must be possible to determine, when validating in document order a sequence of elements against a type, for each element in the sequence, without looking at attributes and without looking ahead, which particle in the type declares the element. This basically means that the content model regular expressions must be *one-unambiguous*.

We shall see that the first constraint will bestow every XML Schema instance with some properties that are very useful for our XSLT analysis, while the second constraint is useless for our purposes (and is indeed not even necessary at all for efficient parse-time validation of XML instances):

A *single-type* regular tree language is a regular tree language, for which for any (q, σ):

$$e^i_\sigma, e^j_\sigma \in m(\delta(q,\sigma)) \Rightarrow i = j$$

The EDC constraint makes XML Schema single-type, under our assumption that type derivation is not used, and with one exception: Wildcards. It is possible in XML Schema to define constructs like:

```
<sequence>
  <element name="a" type="integer"/>
  <any namespace="##targetNamespace"/>
</sequence>
```

Here, the first element declaration will be represented by the regular expression a^i for some i, whereas the latter will, in our terms, be represented by $(e^1_{\sigma_1}|e^2_{\sigma_1}|e^{\cdots}_{\sigma_1}|e^1_{\sigma_2}|e^2_{\sigma_2}|e^{\cdots}_{\sigma_2}|\cdots)$, that is, any name (limited to the target namespace in this example) associated with any type, *among those* $a^j, j \neq i$. For our purposes however, which will be to type nodes statically in order to estimate the possible outcome of an XSL transform run on the node, we will give up any analysis on the results of transformations of wildcards, and just approximate the outcome to "any well-formed XML whatsoever".

Ancestor-language typeability

We want to show that: For all accepted tree state assignments of a single-type regular tree language:

i. For any DNT q, the set of possible ancestor strings of nodes typed q is a regular language.

ii. For any DNTs q_1, q_2, if $q_1 \neq q_2$, then the language of possible ancestor strings of nodes typed q_1 is disjoint with that of nodes typed q_2.

Single-type languages let us determine the DNT of a named *child* element of a DNT, given the child's name and the parent's DNT:

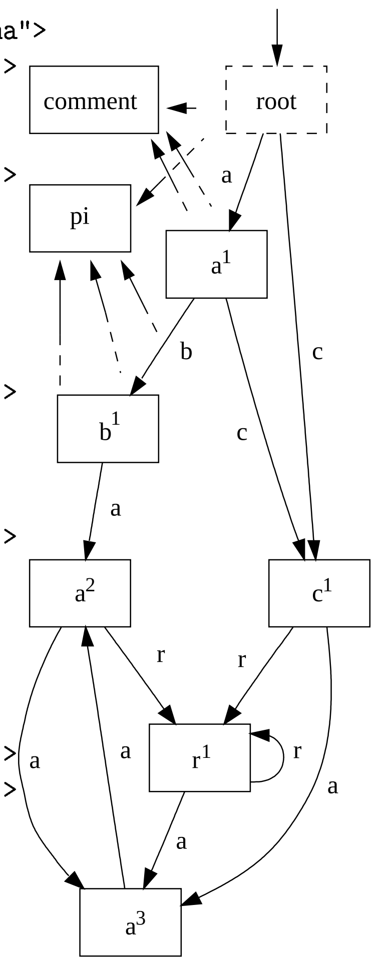
$$\begin{array}{lcl} \delta^{single-type}_{child} & : & \Sigma' \times \Sigma \rightarrow \Sigma' \\ \delta^{single-type}_{child}(q,\sigma) & = & e^i_\sigma : e^i_\sigma \in m(\delta(q,\sigma)) \end{array}$$

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
<element name="a">                    <!--a^1-->
  <complexType>
    <choice>
      <element name="b" type="t1"/>  <!--b^1-->
      <element ref="c"/>
    </choice>
  </complexType>
</element>

<element name="c" type="t2"/>        <!--c^1-->

<complexType name="t1">
  <sequence>
    <element name="a" type="t2">     <!--a^2-->
  </sequence>
</complexType>

<complexType name="t2">
  <choice>
    <element name="a" type="t1"/>    <!--a^3-->
    <element name="r" type="t2"      <!--r^1-->
       minOccurs="0"/>
  </choice>
</complexType>
</schema>
```



An XML Schema with type recursion. The comments contain the DNT names used. Elements named `a` are declared once globally, and twice locally — the global declaration uses a local type definition, and in that, an element reference is used for the `c` element. The other element declarations refer to their type by its name. Since the DNTs `a`^{1} and `c`^{1} are global, either `a` or `c` can be the document element of documents valid wrt. this schema. In the flow algorithm presented later, we assume the one declaration is chosen as *the* document element declaration, and edges from **root** to the others are removed.
Cycles in the DFA indicate that the schema's instance trees have unlimited height (it is type recursive), whereas nothing in the DFA reflects that a choice compositor was used in `a`'s content model, but sequence was used in the others. The `minOccurs=0` cardinality on the "r" particle leaves no trace of itself in the DFA, either, but without that, there would be no finite instance document of the schema (and if such a no-instance schema is fed to a working static validator, any validation result is correct — no instances at all means no invalid instances).
The ancestor languages of the DNTs are:

$anl_d(a^1) = \texttt{a}$	$anl_d(a^2) = \texttt{a(ba\|cr}^*\texttt{aa)(1r}^*\texttt{aa)}^*$
$anl_d(a^3) = \texttt{a(c\|ba)r}^*\texttt{a(ar}^*\texttt{a)}$	$anl_d(b^1) = \texttt{ab}$
$anl_d(c^1) = \texttt{ac}$	$anl_d(r^1) = \texttt{a(c\|(b\|c)a(aa)}^*\texttt{)r}^+$

(i): Consider an automaton whose state set is Σ', its input language is strings over Σ, and its transition function is $\delta_{child}^{single-type}$. Obviously, it is a DFA, and the language of input strings that makes the automaton reach any particular state is regular.

(ii): Now consider any two nodes $v_1, v_2 \in \mathrm{Dom}(t)$ in an accepting run of a single-type tree automaton on a tree t, where

$\lambda(v_1) \neq \lambda(v_2)$ and $ans(v_1) = ans(v_2)$

But it has been shown that the state $\lambda(v)$ of any node v is the state of some DFA run on its ancestor string (specifically, the state is $(\delta_{child}^{single-type})^*(ans(v), \lambda(root(t))))$, and that state is reached deterministically. It is thus not possible that $\lambda(v_1) \neq \lambda(v_2)$ while $ans(v_1) = ans(v_2)$. $\square$

Having established that in single-type regular tree languages, nodes may be typed by their ancestor strings alone, we will define an *ancestor language* function that we will use later:

$anl_d : \Sigma' \rightarrow \mathbf{Reg}(\Sigma)$

DTD is single-type, too, for the simple reason that there can be at most one type for an element name: Single-type regular tree languages contain the local-type regular tree languages, and the class is strictly larger (the single-type example above was not local-type).

6.7 Special XML Schema constructs

XML Schema makes a valiant attempt at mimicking inheritance models of object-oriented programming languages, with *type derivation*: Types may be derived from other types by *restriction*, reflecting a smaller class *extension* (the set of phenomena modeled by the class), or by *extension*, reflecting a larger *intension* (the set of modeled properties of each phenomenon of the class). *Subtype substitution* is one of the fundamentals of the object-oriented paradigm, meaning that at any place where objects of a certain class are expected, objects of a derived class are accepted as well.

Substitution groups

XSD has two different yet rather similar features for enabling subtyping, while retaining typeability at the time a parser reaches its opening tag: One is *substitution groups*, which is a way of allowing an element DNT to appear in place of another, keeping the two distinguishable by requiring that they both are declared globally, with different element names. The DNT that may be substituted is called the *head* of the substitution group, and any DNT that may substitute it — including the head itself — is a *member* of the group. Concretely, an attribute `substitutionGroup="`*name-of-head*`"` is added to the declaration of each member. They work transitively; if a member of one substitution group is the head of another, all members of the last group also become members of the first group. The types of the member declaration must be derived from the type of the head in zero or more steps (apparently, this serves to limit the surprise when a substitution group head is substituted — the type is still compatible).

Substitution groups can be desugared away by replacing every reference to a group head by a `choice` compositor over all the members of the group, but doing so would be unfaithful to XPath2's model of element declarations, which includes substitution group affiliation, and uses that in the `schema-element()` function. If insisting on removing substitution groups, the function could be approximated, though.

Derivation by extension

The other major type derivation feature in XSD is *derivation by extension*, which is only possible where the derived type is complex, as only these can have more than a single member. With this feature, for each derivation by extension of a complex type t, all DNTs declared with type t implicitly spawn a new element DNT, with the *same* element name and with the derived type, and with an implicit declaration of a required attribute, `xsi:type="`*name-of-type*`"`[19] for distinguishability. One consequence of these implied DNTs is that attributes must somehow contribute to element naming, in order for us to remain single-type. Later, when relying on ancestor-language typeability of elements, we will have to find a solution for that.

Types can be *blocked* for derivation by using a `block` attribute on their definition; the effect is to switch off implied subtype substitutability for those types. Derivation of new types from blocked types is still possible, though.

An extension y of a content model x is itself a content model, and it is combined with the extended content model x simply as $\mathsf{Seq}(x, y)$, not exceedingly faithful to the usual idea of object-oriented classes, where properties are named, not ordered. [14] has advised against using extension at all (and against many other features of XML Schema), and use model groups instead to obtain the same effect, without the complications of automatic declaration generation and `xsi:type`.

An upper bound for the set of DNTs mentioned in the content of a DNT, without knowledge of the presence or value of its `xsi:type` attribute can still be made, given a schema:

$$\begin{array}{lcl} W_d & : & \Sigma' \rightarrow 2^{\Sigma'} \\ W_d(q) & = & m(C_m(q)) \cup \bigcup_{q' \in \{\text{all types substitutable in one derivation step from } q\}} W_d(q') \end{array}$$

— which is a set of all DNTs mentioned in the transitive closure of derivation by extension, and certainly an upper bound. The function C_m simply extracts the content model of a DNT.

[19]the namespace prefix `xsi` (or whatever prefix is used instead) must be bound to the namespace `http://www.w3.org/2001/XMLSchema-instance`.

Other derivation features

Derivation by restriction is not quite as hard to deal with as extension. It is simply a way of restricting a content model (replacing it by a sublanguage), a data model (in the same way), or an attribute model (replacing `Optional` cardinalities by `Required` or `Prohibited`), in order to make validation more critical. It can indeed be soundly ignored (as we will do) when estimating an upper bound of for the language of a schema, and its transform: If a DNT (element or attribute) uses a type that is derived by restriction, its set of permitted values is a subset of that of the derived-from type, and the cost of widening is then just a small loss of precision. For simple types, however, that are all derived by restriction from a simple "ur-type" (the type of **char***), restriction *will* be considered.

The *nillable* feature, when used on a complex-typed DNT, permits a so declared element to have no content at all, if it instead has an attribute `xsi:nil="true"`. Like derivation by extension, this feature breaks ancestor-string typeability by effectively declaring a new, empty-content DNT for the original ancestor language. As with derivation by restriction, as long as one is satisfied with upper bounds on DNTs mentioned in a content model, this feature can be ignored.

Abstract element declarations are element declarations that can not be instantiated, only substituted. They are very useful for simulating a context-free specification of s schema; substitution group heads taking the place of nonterminals, with non-head group members as terminals.

6.8 Attribute and data types

Attributes in XML have no order, and in DTD and XML Schema, attribute models can be adequately described as just a set of attribute uses for each element declaration.

In general, an attribute declaration associates a name with a datatype, and an *attribute use* (our own term) associates an attribute declaration with an element declaration, a cardinality, possibly a default value and possibly a statement that the attribute value is fixed to the default value.

In DTD, attribute declarations and attribute uses are wound up in one:

`<!ATTLIST` **element-name (attribute-name attribute-type default-declaration)*** `>`

although in some cases, entities (a simple macro definition and expansion mechanism) is used for repeated attribute uses (explaining how DTDs can contain many more elements and attribute declarations than appearing in the schema text!). **attribute-type** is one of eight regular types defined in [5], and the **default-declaration** contains the information determining whether the attribute is `Required`, `Optional` or `Fixed` (and to which string). In XML Schema, attributes may be declared globally, and referred to from a type definition, assigning a cardinality and a default/fixed value for the reference only.

We will still prescribe ancestor languages for attributes, though, as they we will be needed later. To allow for some experimentation in balancing performance vs. precision, attribute uses were designed as being able to substitute attribute declarations. We first need to extend Σ beyond element names, and Σ' beyond element declarations:

$$anl_d(a^i_\sigma) = anl_d(P_d(a^i_\sigma)).\sigma$$

where P_d is a function $\Sigma' \rightarrow 2^{\Sigma'}$ that, for an attribute use, returns the single element declaration that uses it, or, for an attribute declaration, returns all the element declarations that use it, and anl_d is extended to return, for a set $Q \subseteq \Sigma'$, the union language $\bigcup_{q \in Q} anl_d(q)$.

As with attribute models, DTD and XML Schema are not very expressive in how contents and data can combine. Both have no way of imposing any particular order or dependency between the two, enabling a complete separation of the data model from the content model, as just a function returning a regular sublanguage of $\mathbf{char}^*$.

In already stated, DTD only has a few predefined data types that can describe attribute content, whereas character data in elements is "all or nothing": Either any XML string at all, mixed with declared elements, or no character data at all.

XML Schema offers a much more fine-grained control of datatypes than DTD: Any simple type can be used, and there are more than 40 of them built into the language; all regular or approximately regular [20]. XML Schema has some limitations in its ability to describe mixtures of element and data contents: It can restrict data contents of elements only if the declaration is simple (declares no element content). If the element declaration is complex, data content declaration is "all or nothing", like DTD: Either any XML string content at all is declared as data content (the `mixed` attribute of the declaration is `true`), or no data content at all is declared (`mixed` = `false`).

Again, we can get away with using a simple function (D_d below) on declarations that returns a regular expression over **char** that are declared as a node's data content (or the empty language, if no data content is declared).

XML Schema has a declared wildcard node type for attributes, `anyAttribute`. It represents any number of attributes, with any name and any content, and like `any`, it is to some degree possible to limit the allowed namespaces of the represented attributes. Also like `any`, the problem of fitting it into the schema model is circumvented by leaving it out of the ancestor language model, and let it be accessible separately: We need not model its effect on schema structure, only its effect on transformation results.

[20] An example of a type needing approximation is bounded floating-point numbers: Such a number can be written 1e0, 10e-1, 100e-2 etc., and the "pumping lemma" can prove that languages over them are not regular

A summary of unified DTD and XML Schema

In the following description of the static analysis algorithms, DTD and XML Schema are unified into one description:

The XML Data Model node kinds **comment** and **pi** (processing-instruction) and are assumed to be implicitly declared, with no content or attributes, and the **char*** data model. The **namespace** node kind is deprecated and unsupported in XSLT2, so it is not declared.

To summarize the functions available for accessing information for a schema d, the following functions all have a set Σ_d of all declarations, implicit declarations included, of a schema as their domain:

$$\Sigma_d = \mathcal{E}_d \cup \mathcal{A}_d \cup \{\mathbf{root}, \mathbf{text}, \mathbf{comment}, \mathbf{pi}\}$$

where $\mathcal{E}_d$ and $\mathcal{A}_d$ are the element and the attribute declarations of d.

We will also supplement the information in both DTD and XML Schema schemas with the declaration of a chosen root element for the schema: Both schema languages have no facility for specifying this, but it is a natural item of information to associate with an XML application, and in any case, the information will be needed[21]. In an implementation, the function can be implemented as an auto-detection, as a program parameter, or by asking the user to select the declaration from a list.

$$R_d : \{\} \rightarrow \mathcal{E}_d$$

The **root** node kind is also declared implicitly, with the content model

$$\begin{aligned}\mathsf{Seq}(\quad & \mathsf{Card}(0, \mathsf{unbounded}, \mathsf{Choice}(\mathbf{comment}, \mathbf{pi})).R_d().\\ & \mathsf{Card}(0, \mathsf{unbounded}, \mathsf{Choice}(\mathbf{comment}, \mathbf{pi})))\end{aligned}$$

- C_m returns the content model of a DNT, as an expression in the M-language.
- C_d returns the set of possible content DNTs of a DNT. For **root**, it returns $R_d(\Sigma_d)$, for elements, it returns the element's declared children, **pcdata** if the element is declared to contain text, and **comment** and **pi** if the element is not declared **EMPTY** in DTD. For other node kinds, it returns the empty set.
- W_d returns the set of possible content DNTs of a DNT, widened to remain valid even if the type of the DNT is substituted by any declared, substitutable type. For non-XSD schemas or non-element DNTs, this is just C_d.
- P_d returns the set of possible parents of a DNT; every node that has the argument as a possible child or attribute, or as a contained **pcdata**, **comment** or **pi**.
- A_d returns the attribute model of a DNT, as a set of DNTs, wildcards not included.

[21]Further analysis is based on the knowledge of that. It could be replaced by a set of candidate root elements, though.

- A_m expressions attribute info in the A-language for an attribute DNT.
- D_d returns character data model of a DNT; it is a regular language over **char**, and, with the limitations in the schema language specifications, it is independent of the DNT's content model.
- S_d^* returns the reflexive transitive closure of the set of declared substitute DNTs for a DNT. Abstract DNTs are removed. For any other DNTs than heads of XSD substitution groups, only the argument DNT is in the set returned.
- anl_d returns the regular ancestor DNT language for a DNT. For each DNT q, ancestor languages of all DNTs that q may substitute through the substitution group mechanism are included in a union language. In the implementation, the language is described as an automaton.

The function T_t takes two types as arguments, and decides whether the second may substitute the first — and returns **true**, if the second argument is an XML Schema type, derived in zero or more steps from the first argument, and no step used in the derivation is `block`ed for the derivation kind used. Otherwise, **false** is returned.

The function $T_d : \mathcal{Q} \times \mathcal{E}_d \rightarrow \{\mathbf{true}, \mathbf{false}\}$ takes the QName of a type and a DNT, and returns **true** if the type referred to by the first argument is replaceable by the type of the second argument, as per T_t, and the first argument is not `block`ed for derivation in its XML Schema declaration. Otherwise, **false** is returned.

The function $T_e : \mathcal{E}_d \times \mathcal{E}_d \rightarrow \{\mathbf{true}, \mathbf{false}\}$ forwards to T_d, first extracting the name of the declared type of its first argument.

Some trivia applies to XML declared node types:

- Elements declared with a global type reference have an extra Optional `xsi:type` attribute declared, with a type of just the QName of the type. All implicit declarations spawned by defined extension derivations of the type are made explicit, and each has an extra Fixed(*name-of-derived-type*) `xsi:type` attribute declared.
- Elements declared `nillable=true` have all declarations repeated twice; once set has the declared content model and an extra declared-Optional attribute `xsd:nil` with a type of the singleton value `false`; the other set with the declared content

  $\mathsf{Card}(0, \mathsf{unbounded}, \mathsf{Choice}(\mathbf{comment}, \mathbf{pi}))$

  and a Fixed(`true`) `xsi:nil` attribute declaration.

  A nilledness function N_d is defined as:

  $$N_d(q) = \begin{cases} \mathbf{true} & \text{if } q \text{ has a required } \texttt{xsi:nil} \text{ attr. declaration with the value } \texttt{true} \\ \mathbf{false} & \textit{otherwise} \end{cases}$$

These attributes guarantee that elements copied as-is from input to output by an XSLT2 processor do not lose required attributes in our static simulation. N_d is for the `schema-element()` XPath2 `element()` item test described later.

7 XML Infoset

The XML Infoset specification provides a consistent description and abstraction basis for use in other specifications:

- It applies to well-formed, but not necessarily valid XML
- It abstracts documents and nodes out of their context: An *information item* is an abstraction of a node, which is given abstract *properties* that may be queried for values relevant for the item: A document information item, for example, has a property named **base URI**, providing a base URI to resolve relative URIs in the document against. Even element information items in the document have the property abstractly inherited.
- Namespaces are supported directly (named nodes have a **namespace name** property)
- It has properties for accessing sub-items of an information item: **children** (child elements of elements and the document element of a document), **attributes** (of an element), and **namespace attributes** (for accessing the `xmlns:...="..."` namespace *declarations*, of elements). In the upward direction, all information items except **document** have a **parent** property (for **attribute**s, the name is **owner element**) for accessing their containing item. This model is equivalent to the XPath2 Data Model (to be presented), except for a few naming conventions.
- It models a document as having a **root** node above the root element (also called the *document element*), equivalently with XPath.
- It canonicalizes different representations of the same thing, such as converting `PCDATA` sections, entities and character references to the characters that they represent. Since XSLT runs on top of the Infoset model, we assume in the following that there are no PCDATA sections, entities or character references left in stylesheets.

XML Infoset does not specify any implementation; it only is an abstracting description of local and context information access.

8 XQuery 1.0 and XPath 2.0 Data Model (XDM)

Basically, the XDM is a restatement of XML Infoset, with a definition of *document order*, a definition of node *types*, and the addition of a type system for values. Types are defined in terms of XML Schema.

The most basic term of the type system is an *item*, which has its name from the information items of XML Infoset, but are higher-level entities: An item is either an *atomic value* (a restricted XML Schema simple type instance) or a *node*. Items have an associated *type*, a *typed value* and a *string value* (as it appears in an XML document), and the type is a map between the two: For example, an element node of type `xs:integer` may have the string value “0042”, and the typed value 42. An instance element of a complex type declaration has a string value that is composed of the string values of its text node descendants (which happens to be the same string value that the XSLT instruction `value-of` would evaluate to, given the element), and the typed value is just the string value.

There are seven *node kinds* in the data model:

Document nodes encapsulate XML documents. We label this node kind as **root**. Document nodes can have any number of element, comment, processing-instruction or text children, as opposed to root nodes in XML documents; this is because XPath2 uses document nodes as containers for temporary trees. Opposite Version 1, XPath2 also supports computations on XML fragments which are not documents.

Element nodes, attribute nodes are as in XML.

Text nodes are models of the text that may appear inside element nodes. No text node in the model represents an empty string, and text nodes never have sibling text nodes. We label this node kind as **pcdata**.

Namespace nodes are the namespace declarations that we saw above, separated from the attribute nodes that they are in the XML1.0 sense. It was found after some experience was XPath1’s data model that namespaces bindings are better modeled abstractly than as nodes, and referencing to namespace nodes has become deprecated in XPath2.

Comment nodes, processing instruction nodes are as in XML.

The *document order* is an ordering of all nodes in a document, which we restate here in a simplified form:

- The root node is the first node.
- Every node occurs before its children and descendants.
- Namespace nodes immediately follow the element node with which they are associated. Attribute nodes immediately follow namespace nodes. The relative order is undefined in XDM, but is stable.
- The relative order of siblings is the order in which they occur in the `children` property of their parent node.
- Children and descendants occur before following siblings.

This ordering is called the document order because it orders nodes in a tree in about the same way as their lexical representations — start tags etc. — are ordered in an XML document.

Every value in XDM and XPath2 is a sequence: A single node is a singleton sequence of the node; a sequence of nodes is itself, and the sequences of sequences that may conceptually

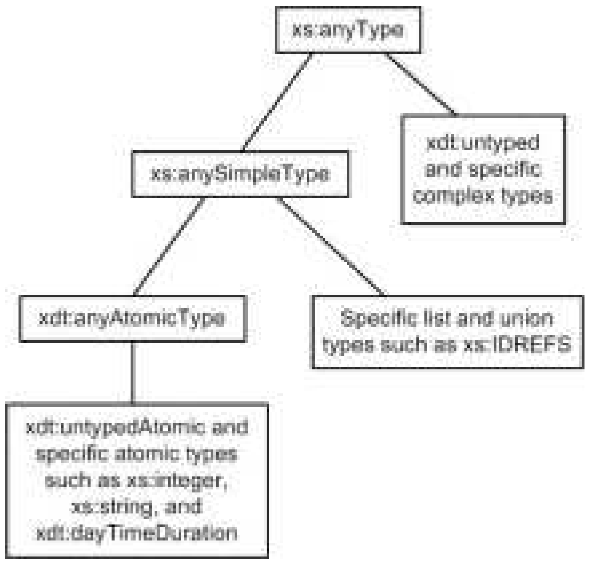

xs:anyType
xs:anySimpleType
xdt:untyped and specific complex types
xdt:anyAtomicType
Specific list and union types such as xs:IDREFS
xdt:untypedAtomic and specific atomic types such as xs:integer, xs:string, and xdt:dayTimeDuration

Figure 13: Simplified XDM type hierarchy. `xs` maps to the XML Schema namespace, and `xdt` to the XDT namespace. **xdt:anyAtomicType** is a non-instantiable supertype of all atomic types. **xs:untyped/xs:untypedAtomic** are for unknown types, as when coming from a document without an associated schema to type it by.

appear in some evaluations are considered flattened and ordered in document order. Sequence types are composed of item types and cardinalities, as what that can be syntactically expressed by XPath2's **SequenceType** nonterminal:

SequenceType	::=	(`empty-sequence()`)
	|	(**ItemType** **OccurrenceIndicator**$^?$)
ItemType	::=	**KindTest**|`item()`|**AtomicType**
OccurrenceIndicator	::=	`?` | `*` | `+`
AtomicType	::=	**QName**

The **OccurenceIndicator** has the usual meaning from regular expressions. Without one, a sequence type becomes a singleton, equal to its item. A sequence is an instance of some sequence type if all its items are instances of its **ItemType**, and its length is within the bounds set by the **OccurenceIndicator**.

The various terminals call for some explanation:

KindTest may produce any of:

`element()`,`element(*)`	Item test for any element
`element(`**QName**`)`	Item test for named elements
`element(`**QName,TypeName**`)`, `element(*,` **TypeName**`)`	Item tests for elements, the type of which are the named type or are derived from it, and not having the `nilled` property
`element(`**QName,TypeName?**`)`, `element(*,` **TypeName?**`)`	Item tests for elements, the type of which are the named type or are derived from it
`schema-element(`**QName**`)`	Item test for schema-declared elements, or their declared substitution-group substitutes
`attribute(`**QName**`)`	Item test for named attributes
`attribute(`**QName, TypeName**`)` `attribute(*,` **TypeName**`)`	Item tests for attributes, the type of which are the named type or are derived from it
`schema-attribute(`**QName**`)`	Item test for schema-declared attributes
`document-node()`	Item test for any document node
`document-node(element(...))`	Item test for document nodes of documents whose document element pass the element test
`document-node (schema-element(...))`	Item test for document nodes of documents whose document element pass the schema element test
`text()`,`comment()`,`node()`, `processing-instruction(...)`	Tests for text, comment, any-kind or processing instruction nodes

`item()` is an item test for any item type (node or atomic type) at all.

The **QName** in the **AtomicType** production are QNames of *atomic* simple types:

There may appear to exist some competition between XDM's sequence types, and XML Schema's list and union-derived simple types: Is a value of a XSD list-derived type a singleton sequence of the list type, or is it a sequence of the list item type? The conflict is eliminated by only permitting XML Schema *atomic* simple types, which are simple types that do not have a list or union step anywhere in their XSD derivation path from the simple ur-type. Thus, only the 19 of the 44 specified XSD simple types that are "primitive simple types"[22] are also atomic types. Instances of non-atomic types, such as IDREFS or TOKENS are converted to sequence types in a process called *atomization*, before they become proper XDM values.

8.1 XPath 2.0

XPath2 is integrated into XSLT2 as a "language in the language"; its primary purposes are selection of sequences of nodes in documents, and output of sequences of items to result trees. XPath2 can also perform computations on values, such as evaluation of boolean and numerical expressions; we will largely ignore this kind of computations, since we concern ourselves mostly with types and not with values.

XPath2 is *strongly typed*, which is good news for static XSLT2 analysis, compared to situation under the dynamic and rather messy type system of XPath1: XPath2 expressions can be type-declared in XSLT2, and with XPath2's optional *static typing feature*, it can be verified

[22] a term defined in the XML Schema Recommendation; they are non-list and nonunion types

that the expressions always produce results that conform to the declared type. Since this verification can be assumed to take place at the time of initialization of an XSLT2 processor, or, if unsupported, that XSLT processing will terminate at type errors , we need not look into it any deeper here.

As an integrated sub-language, XPath2 has a *static context*, as a definition for how static context information is passed to XPath2 from its environment. We will make use of that definition when analyzing XPath, in an assumption that XSLT's imported schemas etc. are known. Parts of the context are:

Statically known documents , mapping document names to document types

In-scope schema types: These are specified to be equivalent with the types that nodes are annotated with during XPath2 evaluation . We assume that the schemas used for static analysis and in the XPath2 in-scope schema types are the same.

In-scope variables: Statically known types of XSLT variables

Function signatures: This component defines XSLT2-defined and XPath2-defined functions that can be called from within an expression. Each function signature can be looked up by name and arity, and contains parameter and return types

Limitations of the XDM type system

The XDM type system is not as expressive as XML Schema content models. As we saw, it can only express sequences of instances of the same type, and only with four different choices of **OccurenceIndicator**, so there is little chance of success in trying to achieve static XSLT validation simply by XDM-typing templates in a XSLT transform (it is possible!) in order to make static typing features validate output. On the other hand, XML Schema content models can express some XDM types, a trick we will use later.

Type annotations in stylesheets will, however, provide us with a chance to make a sharper guess at the possible output of XSLT `copy-of` instructions that invoke a function, or copy the content of unknown, but typed variables or parameters.

XPath2 path expressions

Path expressions are what was known as location paths in XPath1: They select, given an XML tree, a sequence of nodes.

In the XPath2 grammar, almost every expression uses the **PathExpr** nonterminal; even the XPath2 expression `42` is is syntactically a path expression (among many more kinds). We will use the term path expression, written as $\mathbf{XPath}_{path}$ to mean expressions that actually select nodes from an XML tree, in much the same way that the XSLT2 Recommendation does.

A path expression in this sense is a number of *step* expressions, each consisting of an *axis*, a *node test* and a number of *predicates*; for example, the path expression

```
child::foo[descendant-or-self()::node()/child::baz]/attribute::attribute(bar)
```

has two steps; the first uses the `child` axis, and has a `foo` name (node) test and a predicate; the second step uses the `attribute` axis and the XPath2 item test `attribute()`. In this example, both `child::` and `attribute::` could have been omitted entirely, as these axes are the default before name tests and attribute item tests; however, we will write XPath expressions in their full form to avoid confusion. The content of the predicate, `descendant-or-self():node()/child::baz`, could also have been abbreviated to just `.//baz`.

Axes describe directions of navigation in an XML tree, and can be described as functions from nodes to sequences of nodes: A step over the `parent` axis relative to a *context* node evaluates to the empty or singleton sequence of the `parent` property of the node; a step over the `child` axis relative to a context node evaluates to the node's `children` property as a sequence, etc. *Upward* axes like `parent` and `ancestor` arrange the sequence in *reverse* document order; downward steps like `child` and `descendant` in normal document order, and `preceding-sibling` and `following-sibling` in reverse and normal order, respectively. `attribute` orders in an undefined but stable order, and finally, the singleton axis `self` evaluates to a sequence of length 1, and needs not have an order.

Node tests are derived from the **NodeTest** nonterminal. They are the **ItemTest**s already introduced, or **NameTest**s, which are just **QName**s or the token `*`. A node test can be considered a function depleting a sequence, by removing all nodes that fail the test. For **ItemTest**s, the semantics of the test is as in the **ItemType** definitions; for **NameTest**s, a node passes if it has a `name` property (element and attribute nodes have) and the **NameTest** is either `*` or the same qualified name as that of the node.

Predicates are a kind of further node tests, consisting of an XPath2 expression (the full language can be used) enclosed in `[` and `]` and following a node test or a predicate[23]. The expressions in a predicate have access to the functions `current()` and `last()`, which are sources of much confusion: They return the position of a node in a the sequence that is being predicate tested, and the length of that sequence, respectively. That sequence is, in turn, the sequence of the nodes that were left after the node test or predicate immediately preceding the predicate, *not* the one after the axis step:

`child::foo[position() &gt; 1][position() &lt; 3]`[24]

evaluates to up to two nodes: The might be any number of child nodes named `foo`; the first

[23]In Path2, predicates can also appear in some non-path expressions; we will not need to consider that.
[24]&gt; is ">" (greater than) escaped, and &lt; is "<" (less than).

predicate removes the first, and makes a new 1-based sequence of the others. The second predicate retains the first 2 of these.

```
child::foo[position() &gt; 1 and position() &lt; 3]
```

evaluates to at most one node; the second child node of the context node named `foo`. It could have been written as `[position()=2]`, or just `[2]`, which means the same.

Multi-step path expressions have their first step evaluated like a single-step expression, resulting in a sequence. The second step is then evaluated, with each node in that sequence as the context node, and the resulting sequences are merged together in the order given by the step's axis; the remaining steps proceed the same way.

Path expressions can be combined in XPath2 as p_1`intersect`p_2, and p_1`except`p_2, in addition to (and binding stronger than) XPath1's $p_1 \mid p_2$.

Our final word about XPath must be on its size and complexity: The XPath Version 1 Recommendation([6]) and the XML Infoset Recommendation([7]) together are 1449 lines long when pasted into a text editor. The XPath2 Recommendation ([12]), XQuery 1.0 and XPath 2.0 Functions and Operators ([17]) and the XDM Recommendation ([8]) together are 14664 lines; about 10 times as large.

9 XSLT2

The version 2.0 Recommendation of XSLT (currently in Working Draft status) is a significant change from the current XSLT1 status. It has more expressive power, making static analysis more complicated. On the other hand, it is strongly typed, making some aspects of type analysis of the language and its output easier than in the Version 1 generation: Static typing has been thought into the design of XPath2, and static typechecking of its expressions can be safely left to a stylesheet processor that supports it.

The XSLT2 Candidate Recommendation pasted into a text editor is 11283 lines long; the Version 1 Recommendation only 3497.

We will try keeping it below eleven thousand lines here, introducing only major changes from Version 1 to Version 2:

XSLT is a declarative language describing *stylesheets*; programs that will transform XML trees to other XML trees or even non-XML output.

9.1 Stylesheet structure

The most important constituents of a stylesheet are:

- *template rule* declarations
- *named template* declarations
- *stylesheet parameter* declarations
- `key` declarations
- `include`/`imports` declarations
- `attribute-set` declarations
- (in XSLT2): `import-schema` declarations
- (in XSLT2): `function` declarations

`key` and `attribute-set` are not detailed here, but it is described how they are handled in our analysis in Section 12.

Templates and patterns

Template rules are the core components of XSLT: Each has a `match` *pattern* associated, constraining the set of nodes to which the rule applies, and each has a *priority* used in determining which rule wins, in cases where more than one applies.

Patterns are XPath2 path expressions, used not for selection of nodes but for determining whether a *context node* is accepted by, or *matches* the pattern.

Some restrictions from general XPath2 expressions apply:

- Only the `child` or `attribute` axes are permitted
- The special operator `//` is permitted:
  - before the first step of a pattern, where it translates to `fn:root(self::node()) treat as document-node()/descendant-or-self::node()/`
  - between two steps, where it means `/descendant-or-self::node()/`

The semantics of patterns has been changed in XSLT2, compared to earlier versions, to allow for matching on XML fragments that are not rooted in a document node, and for the new XPath2 item tests:

- Patterns with `document-node(...)` as their first node test default to the `self` axis in that step.

- Patterns that begin with a `child` axis step now accommodate for *parentless* content (element, **comment**, **pi**) by automatically converting to the pseudo `child-or-top` axis: That axis has `child` semantics for parented nodes, and `self` semantics for parentless nodes.
- Patterns that begin with an `attribute` axis step now accommodate for parentless attributes, in the same way (`attribute-or-top`).

The semantics of a pattern is now:

> To determine whether a node N matches the pattern p, evaluate the expression `root(self::node())//(`p`)` with N as the context item. If the result is a sequence of nodes that includes N, then node N matches the pattern; otherwise node N does not match the pattern.

It can be seen that *some other node* is involved in the pattern expression; that node is somewhere in between the root of N's XML tree and N itself, or equal to either. In any case, it must exist in the same XML tree fragment as N.

Patterns can not use the XPath2 `intersect` and `except` operators. We will use $\mathbf{XPath}_{pattern}$ to denote any pattern.

Named templates are like in XSLT1. Both template rules and named templates contain a *sequence constructor*, formerly known as a template body: A sequence of *instructions*, defining output and onwards control flow of the template.

Variable-binding elements

The semantics of variables and parameters is one important change in XSLT2: XPath2 *can* evaluate expressions in the context of variable-bound XML trees. XSLT1, with its *Result Tree Fragments*, could not.

It is thus possible to do define, for example, a two-stage transformation[25]:

```
<xsl:stylesheet
  version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:import href="phase1.xsl"/>
<xsl:import href="phase2.xsl"/>
```

[25]The example comes from [15]

```
<xsl:variable name="intermediate">
  <xsl:apply-templates select="/" mode="phase1"/>
</xsl:variable>

<xsl:template match="/">
  <xsl:apply-templates select="$intermediate" mode="phase2"/>
</xsl:template>

</xsl:stylesheet>
```

PARAMETERS

In XSLT2, stylesheet and template parameters may be declared `required`, forcing a caller of functions or templates to bind a value for the parameter. In XSLT1, unbound parameters defaulted to a declared default value, or the empty string if absent.

XSLT2 supports automatic forwarding of value bindings for a template parameter that has been declared as a *tunnel parameter*: Even after flow of control through templates that do not declare a tunnel parameter, it is preserved. A tunnel parameter named p is created by adding a `tunnel="yes" name="`p`"` attribute to a `with-param` instruction; it can be overridden (re-bound) for later template recursion (until the next re-binding, that is) by another explicit `<with-param tunnel="yes" name="`p`" select="..."/>`[26] instruction, or shadowed for a single template invocation in the same way, but without the `tunnel="yes"` attribute.

Tunnel parameters are conceptually similar to dynamically-scoped variables in some functional programming languages, and they are a curse to analyze statically. They make parameter analysis more confusing, but actually not worse. We will chew through the whole story on template parameters in Section 12.

STYLESHEET FUNCTIONS

XSLT2 supports user-defined functions, with parameters and result types defined as **SequenceType**s. In the function definition, an XPath2 expression or an XSLT2 sequence constructor computes the function's value. Functions and named templates are not very different entities; parameters of functions are always `required`, and only functions can be prescribed as just an XPath2 function. Functions, however, output to anywhere whereas templates output to a result tree.

[26]Of course, this also goes for values constructed by a sequence constructor in the `with-param` instead of the `select` expression

Type declarations

Templates, stylesheet functions and stylesheet and template variables may be type-annotated in XSLT2, with an `as="`**SequenceType**`"` attribute, thus strongly typing them[27].

Output validation

XSLT2 features dynamic schema validation of its output, making it possible to specify an XSLT2 transform that either outputs valid XML, or fails at runtime. This could seem like making static validation redundant, then, as output validity can be guaranteed. However, static validation can still make a stronger guarantee; namely that no run-time failure because of an output validation error will happen.

The result construction semantics is bottom-up in XSLT2, a change from XSLT1's top-down. This makes dynamic result validation difficult, because nodes are typed, and as we saw in Section 6.8, the type of a node can be derived from its ancestor string — but at the time of dynamic validation, there is no ancestor string! Therefore, there are provisions for stripping off and redoing type annotation, and for validating against type definitions — not against declarations, as it is usually done.

An obscure aspect of validation is PSVI annotation: Validating trees may add nodes to them. If a tree is recycled in a two-stage transformation, validating or not may change semantics of the 2nd pass, even without validation errors[28].

Grouping

One often recurring problem in XSLT1 stylesheet authoring was, as we saw in the Usenet mini-survey, that of grouping a set of input nodes by some criterion, and process the groups sequentially. The `for-each-group` instruction in XSLT2 was designed to mitigate that. Its format is:

`<for-each-group select="`**XPath**`"` *g*`="`**XPath**`">`*body*`</for-each-group>`

There are some different grouping methods *g* available, with different semantics. We refer to [15].

A sequence of nodes is selected by the `select` expression, and for each node in the sequence, a grouping key is generated by evaluating the *g* expression with the node as the context item. A group is then a largest sub-sequence of nodes that all generated the same key (by the `eq` comparison semantics), and the sequence constructor *body* is invoked for each group, adding to a final sequence that is the result of the `for-each-group` instruction.

[27] See 9.2: Basic stylesheet processors do not understand types, and instead are specified to fail.

[28] Someone ought to specify a something that separates validation and validation-related transformation tasks. Next project...

Inside *body*, the function `current-grouping-key()` returns the current key if g defines one, or the empty sequence otherwise. `current-group()` returns the current group as a sequence.

`apply-imports` and `next-match`

The `apply-imports` instruction works like `<apply-templates select="."/>`, except that it only propagates flow to template rules in stylesheet modules that are imported, directly or indirectly from the stylesheet level that contains the instruction. Its semantics resembles of a call to an overridden method in an object-oriented language, from within the overriding method: `super.foo()` in the body of `foo()`, and the purpose is the same: To allow programmers to specialize some behavior without having to repeat the prescription of the general part.

The `next-match` instruction works the same way as `apply-imports`, except that control may go only to template rules with a lower import precedence than that of the containing rule (which is different from `apply-imports` — a stylesheet level may have a lower precedence than another, without having to be imported from it!), *or* to template rules with the same import precedence and a lower priority than that of the containing rule.

Built-in template rule model

The built-in template rules in XSLT2 are rather similar to those in XSLT1: Output the `value-of` any text and attribute nodes, recurse to templates matching all children of element and **root** nodes, and do nothing for **comment** and **pi** nodes. The built-in rules in XSLT2 now have a `mode="#all"` added, no longer necessitating conceptual on-the-fly creation of default rules for each mode used. The rule for element nodes also forwards the current mode as #**current** and any template parameters, as opposed to the one in XSLT1 which shed the parameters.

This behavior actually makes built-in rules quite easy to handle; only the implicit parameter forwarding needs consideration. Again, the treatment of this goes into Section 12.

9.2 Conformance and restrictions

The *Conformance* part of the XSLT2 Recommendation specifies two different levels of stylesheet processors (programs that execute XSLT transforms) conformance:

Basic XSLT processors do not understand schemas, and enforce type safety by failing at any `import-schemas` instruction or `as` type declaration with other than built-in types in a stylesheet.

Schema-Aware XSLT processors can import schemas; they enforce typing of type-declared variables and parameters, and they can dynamically validate their output against a schema.

10 XSLT2 AND STATIC ANALYSIS

The XSLT2 changes that affect the static analysis situation, compared to Version 1, are:

Functions: In addition to stylesheet-declared user functions, the library of predefined XPath2 functions has expanded significantly from that of XPath1, meriting a whole Recommendation of its own ([17]).

Variable-binding Constructs: As we have seen, XPath2 can perform evaluation on variable-bound XML trees.

Grouping: `for-each-group` is handled in Section 12.

Value-of: The text-node constructor instruction `value-of` now accepts and outputs sequences, not only items[29]. It has also been extended by a `separator` attribute for specifying separator strings; this needs to be considered. `value-of` has no implications for flow; its handling in validation is explained in Section 14.4.

Modes: XSLT2's polyvariant modes complicate flow analysis somewhat. This is described in Section 13.4.

`apply-imports`/`next-match`: The way we model these instructions is described in Section 13.5.

Initial contexts: In XSLT2, an initial context node and an initial named template name or an initial mode can be supplied from the environment of the stylesheet processor (see [15], Section 2.3). Since none of these poses any threat, or challenge, to the feasibility of our static analysis, and we found no examples of transforms using the feature, these have been fixed to default values: `/` for the initial context node, #**default** for the mode, and no initial named template name. This reflects XSLT1 semantics.

Multiple input documents: XPath2's static context definition makes it possible to statically retrieve any schemas associated with documents loaded with the XPath2 `doc` function, and the XSLT2 `document` instruction. These both evaluate to the root node of the document loaded. Semantics of XPath expressions are defined such that information about which schema it applicable can be reached from every node, so multiple documents are no special challenge in static analysis. To save the complications, we will not pursue multiple input documents any further.

Multiple output documents: XSLT can create and output any number of XML trees in each transform. Since instruction for output of other than the primary result tree can

[29]In XPath1, `value-of` will drop any but the first item, if its argument is a node-set

statically associate the tree with a schema, statically validating for more than one result tree is only a matter of retrieving the relevant schema, and process as for the primary result tree. This is not pursued any further here.

Pattern semantics: The troublesome pseudo-axes in XSLT2 patterns will not affect us, as they have effect only for parentless nodes in other patterns than `/`, that is, for variable-bound nodes. We treat variable-bound nodes by adding a pseudo-parent to them, making the pseudo-axes redundant.

Other changes will not need any consideration in this work, including:

Value computations, other than `value-of`: A large number of improvements have been made in expression evaluation. Semantics of relational operations are defined in the XML schema value spaces in XSLT2, making different representations of the same typed value identical, as opposed to XPath1's use of its own type conversion for operators. XPath2 also features new quantifier expressions ("for any" and "for all")[30]. In the present version of our analysis, which makes no attempt at all to statically bound the results of other than a few string expressions, the new operators are, like the XSLT1 ones, ignored.

Serialization: Features for finer grained result tree serialization have been added, whitespace semantics was refined, and `disable-output-escaping` has become deprecated and replaced by character maps. We have ignored this, as our analysis operates at the tree level of XML, not considering serialization.

Dynamic errors: Many previously ignored erroneous situations have become defined static or dynamic errors. Since a non-recoverable dynamic error raised guarantees against invalid output of a transformation (stopping it), we have ignored this aspect. Of course, it is also interesting to make guarantees that this cannot happen; static typechecking is specified as an option for XPath2, and as implementation-defined for XSLT2.

Part III

XSLT flow analysis

11 Definitions and restrictions

We assume in the following that a schema for any XML that is used for input for an XSLT transformation is available to our analysis, and, when a schema-aware stylesheet processor

[30] As a curiosity, the claims in the first four use cases in Section 1.3 of the XPath2 requirement document (`http://www.w3.org/TR/xpath20req/`) about comparisons that XPath1 can not express — are all wrong!

is used, that any schemas declared with `import-schema` are those that will be used in our analysis. For basic stylesheet processors, we need to assume only that they are compliant, ie. never output anything from stylesheets that use schema typing features for other than primitive datatypes.

For XSLT2, some language constructs (that are not really core features, and are not a challenge to the basic feasibility of our algorithm) have been ignored completely. They are:

- The `exclude-result-prefixes` standard attribute
- `analyze-string`, `matching-substring`, `non-matching-substring`
- Conditional element inclusion; `use-when`
- Backwards and forwards compatible processing
- Extension functions, extension instructions and the `fallback` instruction
- Collations, and their meaning
- XPath2's statically available collections and collations, and their defaults
- Character maps
- Result tree serialization, `disable-output-escaping` and non-XML output
- User-defined data elements
- `format-number` formats, `system-property`, `element-available`...

Below-XDM model details like expansion of entities etc. are abstracted away at this level.

We assume that the transform outputs XML, not HTML or text. HTML transforms can be analyzed after XHTML conversion, though.

Output and onwards control flow from templates that match **comment** and **pi** nodes are *not* considered, because these node kinds are not declared, and are allowed almost anywhere in an XML document. In cases where **pi** nodes are matched for output, we can only advise users to wrap them in a freshly declared element, and copy the matching template's content to one matching the new element.

We assume syntactic and referential validity of input stylesheets and schemas, even though the implementation uses schema validation on these.

The algorithm described bases largely on approximation; good approximations again base on good estimates on where to focus particular attention. [23] and [20] both have excellent empirical backing of their basic decisions — and where we have extended upon their work directly, we see no reason to repeat their observations. We refer to the two articles.

Finally, again, transform validity is defined in terms of a stylesheet T, an input schema and an output schema. The output schema is not used until later in the process, but the input schema D_{in} is used so often that we will just call it d, and use that as a qualifier for most function names related to it.

12 Stylesheet simplification

Stylesheet simplification is the process of reducing the complex and somewhat redundant XSLT2 language to a smaller *core XSLT2* language, reducing the number of constructs that need to be considered in later analysis. It is divided into two overall stages:

1. *Semantics-preserving* simplification translates an XSLT2 stylesheet to a core language stylesheet with all semantics preserved
2. *Approximate* simplification no longer preserves semantics, but preserves any validity error there may have been in the original stylesheet

Stylesheet simplification was used in previous work ([23]); our contribution is a definition of a general-purpose implementation of a single-pass process[31], modeling stylesheets more coherently with the XSLT2 Recommendation (import tree structure in particular) and more generally.

Backmapping of desugared constructs was added in the editor-oriented implementation: Each element is decorated with a *source ID*, and a *core ID*, along with parse location information. During all stages of simplification, whenever an element in a source stylesheet module or schema document is converted to a different construct, the original element's source ID and parse location are copied to the replacement construct. In subsequent analyzes, both can be back-mapped to source documents, helping target any message about that particular element.

The primary tasks of semantics-preserving stylesheet simplification are:

- To reduce most control-flow related instructions to `apply-templates`, preserving semantics by adding moded template rules
- To resolve, as far as possible, stylesheet and template variables and parameters
- To unify all branching instructions into `choose` instructions with `otherwise` branches
- To reduce all text output to `value-of` expressions

First, we will introduce a new design of ours, for replacing XSLT instructions whose *effect on control flow* or *effect on output* depend on a typed value.

[31]Its implementation is much faster than that of [23], but considerable time was also spent designing it!

12.1 Type reconstruction

A few extra function definitions are used:

- `xslv:unknownCollation()` is a pseudo-collation with an unknown ordering.
- `xslv:unknownSequence()` is a function evaluating to anything in `item()*`.
- `xslv:unknownBoolean()` is a function evaluating to an unknown boolean value.
- `xslv:unknownText()` is a function evaluating to any string in **char***.

Type reconstruction is a simple idea[32], helping to reduce the two type systems of XPath2 **SequenceType**s and the content models of schema declarations to just one: **SequenceType**s are translated to content models where possible.

Assume that i is a `copy-of`, `value-of` or `apply-templates` instruction, with a `select` expression:

`<`i `select="$`pr`"/>` or `<`i `select="fn:foo"/>`

where

- the variable reference p or the function `fn:foo` is declared to have a sequence type of type t with **OccurenceIndicator** o
- the path expression r is empty, or begins with / otherwise

`copy-of`, `value-of` output things, depending on a value typed as a sequence of t, and `apply-templates` controls flow, with the same dependency. The instructions can be approximated:

if t **is** `element(...)` **or** `schema-element(...)`**:** An element declaration x is added to a fresh schema, with no attributes or data content:

$$C_m(x) = \begin{cases} y & \text{if} \quad o \text{ is absent} \\ \mathsf{Card}(0, \mathsf{unbounded}, y) & \text{if} \quad o = * \\ \mathsf{Card}(1, \mathsf{unbounded}, y) & \text{if} \quad o = + \\ \mathsf{Card}(0, 1, y) & \text{if} \quad o = ? \end{cases}$$

Another element declaration y is added to the fresh schema, with no attributes or data content: $C_m(y) = \mathsf{Choice}(T)$

where T is an upper-approximated set of element DNTs that pass the item test t; the computation of this is described in Section 13.8.

[32] Proposed by us; currently not implemented fully.

An `import-schema` declaration, referring to the fresh schema, is added to the stylesheet module being simplified.

The following template rule is also added to the module:

`<template match="xslv::`$\mu(x)$`">`
`<`i` select="child::`$\mu(y)$`/child::*`r`"/>`
`</template>`

and the original instruction i is replaced by

`<apply-templates select="xslv::`$\mu(x)$`">`

The axis `xslv` is a pseudo-axis introduced, defined to map to only the element declarations added here. Since these declarations are not reachable from the defined schema document element, they do not interfere with the schema-based analysis that we will soon introduce.

if t is `attribute(...)` or `schema-attribute(...)`: An element declaration x is added as in the previous case. Another element declaration y is added, with no element or data content: $A_d(y) = A$

where A is the set of attribute DNTs in the input schema that pass the item test t.

The following template rule is added to the stylesheet module being simplified:

`<template match="xlsv::`$\mu(x)$`">`
`<`i` select="child::`$\mu(y)$`/attribute::*`r`"/>`
`</template>`

and the original instruction i is replaced by

`<apply-templates select="xslv::`$\mu(x)$`">`

if t is `text()`: The original instruction is replaced by `<`i` select="xslv:unknownText()"/>` if i was `xsl:value-of` or `xsl:copy-of`, otherwise by
`<apply-templates select="/descendant-of-self::node()/text()"/>`

if t is an AtomicType: If i is `apply-templates`, there is a static error in the stylesheet; report it and terminate. Otherwise,

An element declaration x is added to a fresh schema, as in the first case. Another element declaration y is added, with no element content or attributes: $D_d(y) = \mathcal{L}(t)$

that is, y is a simple-typed element declaration with the **AtomicType**.

The following template rule is added to the stylesheet module:

`<template match="xslv::`$\mu(x)$`">`
`<`i` select="child::`$\mu(y)$`/child::text()`r`"/>`
`</template>`

and the original instruction i is replaced by

`<apply-templates select="xslv::`$\mu(x)$`">`

Any other case: — including t=`item()` and t=`node()`: There is too little type information available to make a proper model. `<copy-of select="xslv:unknownSequence()"/>`

is substituted for $i =$ `apply-templates` or `copy-of`, or `<value-of select="xslv:unknownText()"/>` for $i =$ `value-of`,

and a warning is issued, asking the user to sharpen the declared type if possible.

This extra-declaration hack unfortunately needs a pseudo `xslv` axis, but then again, it can squeeze the most of two type systems into one (and the cases for which it can not are almost hopeless to analyze statically anyway).

12.2 Staged simplification

The implemented semantics-preserving simplifier is modular, and separable from the approximate simplifier and the static analysis part of the code. The semantics-preserving simplifier can be considered a general-purpose simplifier for XSLT, as it significantly reduces the language, without doing any harm.

Semantics-preserving simplification

A number of sequential stages for simplification were identified:

Loading: Stylesheet modules in the simplified XSLT2 syntax are wrapped in a template rule with a `/` match pattern, returning them to full syntax. Stylesheet modules named `transform` are renamed to the equivalent `stylesheet`.

All `include` and `import` instructions in the primary stylesheet module are expanded, and an *import tree*, as of [15, XSLT2, Section 3.10.3] is built, pulling in the entire stylesheet. Each stylesheet module is validated against an XML Schema for XSLT2 by the parser, preventing transient syntactic errors occurring during on-line editing from setting off a confusing plethora of error messages from back-end code. Backmapping information of element IDs and parse locations is added.

Context-free simplification: Each stylesheet module is processed a top-down, non copying manner: All attribute values that are specified to contain XPath expressions are parsed, and the parsed expressions are cached. Expressions at top-level, outside any templates, are simulated as being evaluated in the initial context, rewriting:

- `current()` to the initial context item requested from the validation environment (which defaults to `/`)[33]
- `position()` and `last()` to `1`
- `name()`, `local-name()` and `namespace-uri()` to the results of evaluating the functions on the value substituted for `current()`

[33]Only the default is currently implemented in the demonstrator program

For some instructions, if an XPath attribute is absent (parser defaults insertion turned off), a default is added:

- `child::node()` for `select` attributes in `apply-templates` instructions.
- `#default` for `mode` attributes on `apply-templates` instructions and all template rules.

For some elements, if an `as` attribute is absent (parser defaults insertion turned off), a default is added:

- `item()*` for `as` attributes in `variable` instructions.
- `item()*` for `as` attributes in `param` instructions.
- `item()*` for `as` attributes in `with-param` instructions.
- `item()*` for `as` attributes in `function` top-level elements.

Restructuring simplification: Each stylesheet module is processed a bottom-up, copying manner:

- `<text>`*s*`</text>` elements are converted to `<value-of select="'`*s*`'"/>`, preserving whitespace.
- All adjacent text nodes are joined[34] into a string s, and depending on the applicable value of the `space` property[35] and on whether s is all whitespace, it is discarded or converted to `<value-of select="'`*s*`'"/>`.
- `<if test="`*e*`">`*seq*`</if>` expressions are converted to
  `<choose><when test="`*e*`">`*seq*`</when></choose>`
- `choose` expressions without an `otherwise` branch have one added, with an empty sequence constructor.
- All top-level variables and parameters are bound in the scope of their containing stylesheet level. Variables will later be resolved where they are used, and parameters will trigger[36] a request for a value from the validation environment.
- All top-level attribute sets are bound in the scope of their containing stylesheet level.
- All key definitions are bound in the scope of their containing stylesheet level.
- References to all `call-template` instructions are collected and bound in the scope of their containing stylesheet level.
- All literal elements

  `<`$elename\ att_1 = "avt_1" \cdots att_k = "avt_k"$`>`*body*`</`*elename*`>`
  in a namespace different from the XSLT namespace are converted to

[34]Not all DOMs do this automatically
[35]as derived from the scoped `space` attributes of ancestors, see [XSLT2] Section 4.2
[36]Not implemented in the current demonstrator program

```
<element name="$elename$">
  <attribute name="$att_1$">
    <value-of select="$avt_1^1$"/>
...
    <value-of select="$avt_1^{l_1}$ "/>
  </attribute>
...
  <attribute name="$att_k$">
    <value-of select="$avt_k^1$"/>
...
    <value-of select="$avt_k^{l_k}$"/>
  </attribute>
[the converted body]
</element>
```

where avt_i^j is the jth XPath expression parsed from the ith attribute's attribute value template: The attribute value template (AVT)

`foo{substring-after($clark, '}}')}`
is, for example, parsed to

{`'foo'`, `substring-after($clark, '}')`}.
Attributes *in* the XSLT namespace not named `space` are not converted, though, but are copied to the resulting `element` element constructor.

Variable resolution: Each stylesheet module is processed in a bottom-up, copying manner:

- Variable references, including those in bound attribute sets, are resolved, as far as possible.

  Global and local variable references are replaced by their definitions, except:

  1. References inside a `for-each` instruction that refer to locally-bound variable (in the same template) that transcend the `for-each` instruction, are not resolved. This is to maintain soundness of a later move of the instruction's body to a fresh template rule, forwarding the named bindings as template parameters.
  2. References to variables bound by XPath2's `for` clauses, quantified expressions etc. that declare intra-expression variables. These are pinpointed using standard scoping analysis.
  3. For `variable` declarations containing a sequence constructor, references to these are *only* resolved in one case: `<copy-of select="`e`"/>` instructions, where e is a variable reference, are outright replaced by the content sequence constructor[37].

[37]Expressions referring to the current context item have been replaced by the initial context item already. At this stage, there is also a possibility of removing `copy-of` instructions that have a stylesheet function in their `select` expressions, replacing them by the sequence constructor of the function declaration. However, the only situation where that would be worthwhile is when the function evaluates to a sequence of nodes — we do not expect stylesheet authors to do the `copy-of` functions, then; named templates are the thing to use.

- All instances of `use-attribute-sets`, including those in bound variables, are resolved, taking care that attributes already present override set attributes properly, and that sets override each other in the specified way.
- Each *body* of

  `<for-each select="`e`">`
  sorts
  body
  `</for-each>`

  is copied to a template rule:

  `<template mode="`m`" match="child::node() | attribute::* | /">`
  `  <param name="`v_1`" as="`as_1`"/>`
  ...
  `  <param name="`v_k`" as="`as_k`"/>`
  the converted *body*
  `</template>`

  where m is a fresh mode, $v_i \cdots v_k$ are the `for-each`-transcending variables mentioned above, and $as_1 \cdots as_k$ are their declared types. The `for-each` instruction is replaced by

  `<apply-templates mode="`m `" select="`e`">`
  `  <with-param name="`v_1`" select="$`v_1`" as="`as_1`"/>`
  ...
  `  <with-param name="`v_k`" select="$`v_k`" as="`as_k`"/>`
  sorts
  `</apply-templates>`

Template simplifier: Each stylesheet module is processed in a top-down, non copying manner, replacing named templates, and splitting union `match` patterns:

1. Templates with both a `name` and a `match` attribute are modified to two templates; a rule without the `name` and a named template without the `match`.
2. Named templates have a `mode="`m`"` attribute for some fresh mode m, and a
   `match="child::node() | attribute::* | /"`
   attribute added.
   All
   `<call-template>`
   `  <with-param name="`p_1`">`
   ...
   `  <with-param name="`p_n`">`
   `</call-template>`

instructions invoking the named template are altered to[38]

```
<apply-templates select="self::node()" mode="m">
  <with-param name="position" select="position()" as="xs:integer"/>
  <with-param name="last" select="last()" as="xs:integer"/>
  <with-param name="p1">
...
  <with-param name="pn">
</apply-templates>
```

The `names` of the templates are then removed.

3. All template rules have

```
  <param name="position" select="position()" as="xs:integer"/>
  <param name="last" select="last()" as="xs:integer"/>
```

prepended to their parameter list[39]. All calls to the functions `position()` and `last()` are altered to become references to variables of the same name, except that `position()` inside `for-each-group` is left unchanged. This serves to hide that `call-template` retains the *focus* (the values of the two functions, and the context item) while the `apply-templates` that will replace it updates it.

4. All `apply-imports` and `next-match` instructions have their focus forwarded through parameters in the same way as `call-template`. This serves to allow the focus-altering `apply-templates` and the focus-retaining `apply-imports` and `next-match` instructions to be treated uniformly later (the latter two are *not* replaced).

Template rules with a union `match` pattern are split into a set of template rules that are identical, except that their `match` patterns each are a non-union subexpression of the union.

Template rules that do not have an explicit `priority` attribute have one added, containing the default priority computed from the `match` pattern of the rule, as by [15], Section 6.4.

Remaining named templates are unused, and can be removed.

Approximate simplification

Key-killing: All function calls to the `key` function with key name k are replaced by the path expression `/descendant-or-self::node()/`$k(k)$, where $k(k)$ is the `match` attribute for the key's definitions, bound in the environment of the stylesheet module one step back. However, if $k(k)$ is absolute, no `/descendant-or-self::node()/` is prepended.

[38]`xs` is bound to the XML Schema namespace, `http://www.w3.org/2001/XMLSchema`

[39]If some of these names are in use as parameter names already, the other names are consistently converted.

For-each-group: Each `for-each-group` instruction:

`<for-each-group select="`p_1`" `g`="`p_2`">`*body*`</for-each-group>`

is translated to:

```
<apply-templates select="p1" mode="m">
  <with-param name="current-group" select="p1" as="node()+"/>
  <with-param name="current-grouping-key" select="p1/p2" as="node()?"/>
  <sort select="xslv:unknownCollation()"/>
</apply-templates>
```

if g is `group-by` or `group-adjacent`; otherwise, p_1/p_2 is replaced by `"empty-sequence()"`. The `sort` serves to indicate that sequences are reordered in some unknown way.

A new template is added:

```
<template match="child::node() | attribute::* | /"  mode="m">
  <param name="current-group" as="node()+" required="yes"/>
  <param name="current-grouping-key" as="node()?" required="yes"/>
body
</template>
```

Inside *body*, function calls to `current-grouping-key()` are replaced by a variable reference to `$current-grouping-key`, and function calls to `current-group()` are replaced by variable references to `$current-group`.

Template parameters: First, let us recapitulate what is left of variables and parameters:

- References to variables bound by sequence constructors and referred to from the `select` expression of `value-of` and `apply-templates` are still present
- Stylesheet and template parameters are still present

There are different ways to proceed with template parameters, in order of growing ambitions:

Leave for later: Not feasible. The later flow and validation analyzes are too difficult to model conservatively with variables and parameters.

Smudge: All template parameter references are replaced by `xslv:unknownSequence()`. This is unacceptable, given that template parameters often end up in output.

Flow-insensitive resolution: All `with-param` instructions are partitioned by parameter name, and all variable-binding expressions for each partition is collected (those that depend on a context node will have to be approximated; they cannot reliably be moved). Every reference to a template parameter with some name is now replaced by a `choose`, ranging over all bindings in the name's partition, with `xslv:unknownBoolean` test expressions. This is the approach suggested by [20], and it was fine for XSLT1, where the only nasty construct, RTFs, could be replaced by `xslv:unknownText()`.

Declared-type based resolution: The XSLT2 `as()` type declaration for the variable of parameter is used, replacing variable references by some other construct, constrained by the declared type

Hybrid type-based and flow-insensitive resolution: This has the advantage over a pure type-based solution that it will retain some precision the case of a pure XSLT1 stylesheets, where variables and parameters are untyped.

Flow-sensitive resolution: If the effect of parameter binding on control flow can be reasonably approximated — it can, if there are no parameter references in any `apply-templates` instruction — then resolution could be performed *after* flow analysis, sharpening the set of possible bindings for each reference.

Hybrid, with inference: As the above, inferring more specific types than declared, where possible, as a luxury for the stylesheet author. For XSLT2, resolution of template parameters can be improved over that over that of XSLT1, because they may be declared `required`, eliminating the default binding to the empty sequence.

Currently, our implementation uses flow-insensitive resolution, because it automatically emulates both parameter tunneling and the parameter-forwarding effect of XSLT2 built-in templates: It is easy. For a real XSLT tool, put into the hands of users who do not know these details, a flow-sensitive approach seems easier to defend.

Function call expressions and template parameters: After variable and parameter resolution, function call expressions still remain as non-path expressions that may affect control flow or output. No attempt is made to statically analyze functions, and stylesheet parameters are not resolved (by asking for a value). Instead, both of these are approximated for `value-of`, `copy-of` and `apply-templates` using our type reconstruction technique, and just left as-is for other instructions.

`<copy-of select="`e`"/>` instructions are converted to:

`<apply-templates mode="`m`" select="`e`">`

and a template rule is added:

```
<template mode="m" match="child::node() | attribute::* | /" priority="0">
  <copy>
    <apply-templates mode="m" select="child::node() | attribute::*"/>
  </copy>
</template>
```

(this rule is re-used for all `copy-of` instructions converted, with the same mode m)

`number` instructions are converted to `value-of`

Scope	`copy-of`	`value-of`	`apply-templates`	other
global, exp	resolve	resolve	resolve	resolve
global, seq	replace	t	t	leave as-is
local, exp	resolve	resolve	resolve	resolve
local, seq	replace	t	t	leave as-is

Table 2: How variables can be dealt with: **exp**ression-declared variables are generally easily resolved. **seq**uence-declared variables can outright replace `copy-of` instructions referring to them by a copy of the sequence, but are harder to deal with for other instructions — we suggest **t**ype reconstruction. In XSLT1, sequence-bound variables were RTFs, and could not be referred to from `apply-templates` instructions. In XSLT2, they *can*.

Scope	`copy-of`	`value-of`	`apply-templates`	other
global	t	t	t	leave as-is
local	resolve+`choose`-wrap	resolve+`choose`-wrap	resolve+`choose`-wrap	leave as-is

Table 3: How parameters can be dealt with: Global parameters can, of course, at best be type-approximated, unless one accepts a simplified stylesheet that only represents one particular binding — in that case, the parameter can be treated as a global variable. Local parameters need resolution and replacement of the referencing instruction by alternatives.

Experience with implementation of stylesheet simplification:

1. Namespace binding scoping makes stylesheet document restructuring painful! To be certain of correctness, it is necessary to parse all XPath expression and QName attributes, and resolve namespace prefixes to URIs. After restructuring, it is again necessary to verify that all namespace bindings are still intact. XSLT2's `xpath-default-namespace` feature (that allows binding of unprefixed QNames in XPath expressions to namespaces) is likewise scoped, and just makes it even worse. A good brute-force technique might be to copy namespace bindings to every node in their scope, before any restructuring is done.

2. Using parameters for forwarding variables and focus is not a problem in semantics preserving simplification, but after approximate simplification, there is an unnecessary loss of precision (which could be avoided by using fresh names for the parameters).

3. It is felt, after the experience of implementing stylesheet simplification, and upgrade it with backmapping, that it is a fine technique for research projects where empirical data are used, helping reduce the number of entities necessary to explain. However, in a production setting where better backmapping and especially better scalability (nested `for-each` expressions desugar in an all-out code explosion) are needed, *abstraction* would probably work better than *translation*: Every construct that evaluates an XPath path expression, given a context (focus), and passes control to some other construct, could be abstracted the object-oriented way, and the particulars — `apply-templates`, `for-each` etc. — could be made as specializations of that. Likewise, sequence constructors could have their own abstraction, with its own context set (see the flow analysis section), and a general abstraction could be made of containers of sequence constructors.

Simplification has the advantage of staying within a well-known language (XSLT2) a long part of the way, but in the present project, implementation of the simplifier was disproportionally difficult, compared to the simplicity of the constructs that benefited from it.

Splitting template rules with union patterns into single-path pattern rules *is* a great ad-

12.3 The core XSLT2 language

We have selected *not* to formally specify a grammar for the core XSLT2 language, because a carving out of every element and every attribute removed or retained would make a very long and rather useless list anyway. Instead, we just assert that the core-language stylesheet:

- Is syntactically correct, less the `xslv` axis.
- Preserves any transform invalidity of the original (and possibly adds more).
- Has had all `call-template` instructions invoking named templates replaced by `apply-templates` instructions, semantics preserved.
- Has had all `for-each` instructions replaced by a semantically equivalent `apply-templates` and template rule construct (namespace nodes are ignored).
- Only has template rules; no named templates.
- Has no union template rule patterns.
- Has instrumentation data identifying cases of template priority conflicts introduced by union pattern splitting.
- Has an explicit priority on all templates. In the templates that were created with a fresh mode, the priority is irrelevant, though.
- Has had all literal text replaced by `value-of` instructions.
- Has had all literal result elements replaced by equivalent `element` instructions.
- Has had all literal result attributes replaced by equivalent `attribute` instructions.
- Has no `copy-of` instructions.
- May have function declarations, but these are not used from `value-of`, or `apply-templates` instructions (except `xslv:unknownSequence()`).
- May have variable declarations, but these are not used from `value-of`, or `apply-templates` instructions.
- May have template parameters, but these are not used from `value-of`, or `apply-templates` instructions.
- Has no `number` instructions — so only `value-of` and `copy` can create text nodes.

hold after simplification. Also, all XPath2 expressions in the stylesheet not mentioned here are retained fully, except that implicit axes have been made explicit.

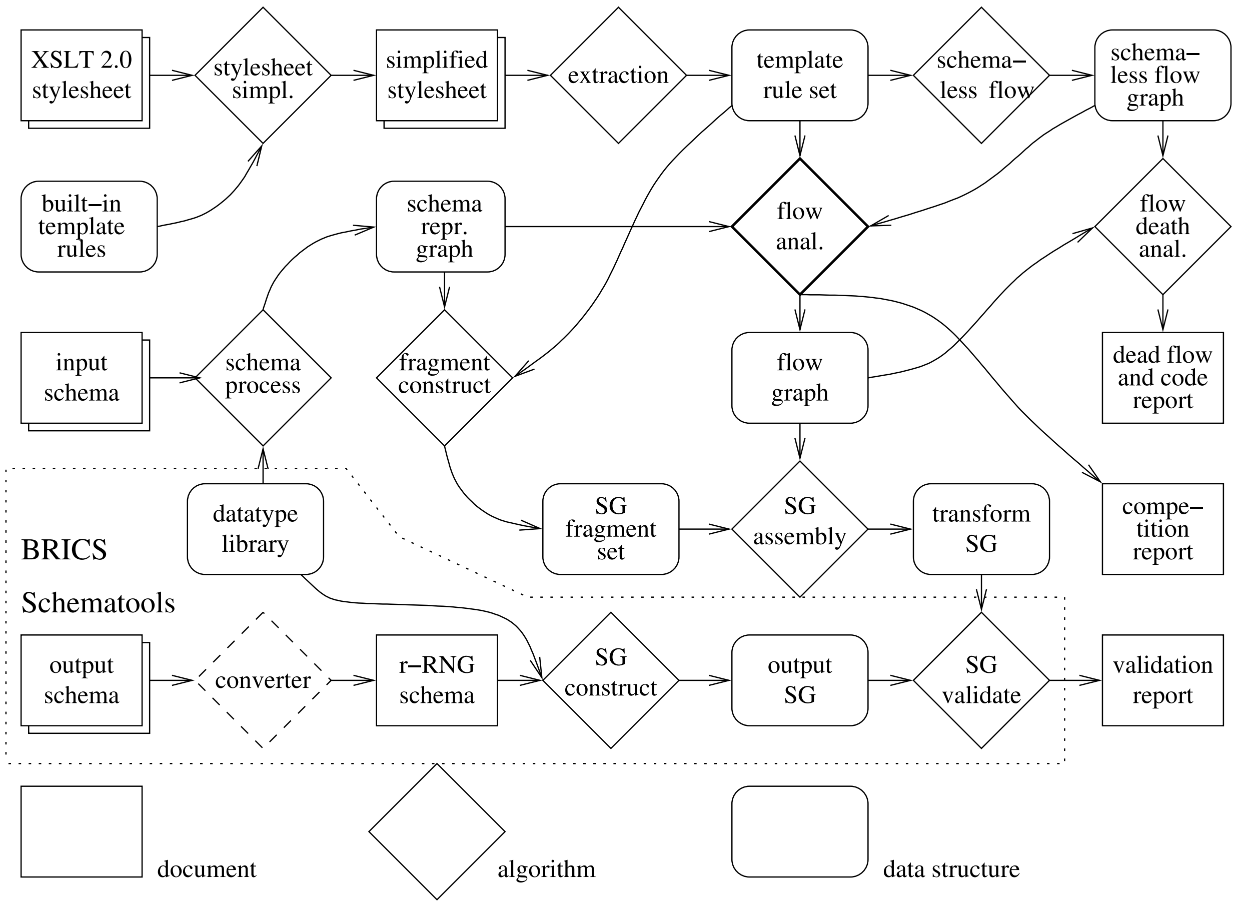


Figure 14: The overall data flow in the exploratory program implemented is rather involved: Input is on the left-hand side of the figure. BRICS Schematools is an external component used for transform validation. The major components are the *flow analysis*, depending on a set of template rule representations and an input schema, and the SG (Summary Graph) construction and assembly, generating summary graphs for BRICS Schematools to validate.
On the right-hand side are the outputs that target fulfilling of our goals: A dead flow and code report containing information on empty-selecting expressions and dead code, a competition report on template priority issues, and a validation report on the overall validity of the transform output with respect to an output schema.

13 Flow analysis

In this part, we will present the algorithm devised as the main engine of our proposed type-safe XSLT tool, the background definitions necessary to understand it, and the results of some experiments performed to verify the design.

13.1 Goal of the analysis

Flow analysis is the computation of a *flow graph.* The analysis is done on basis of a stylesheet and an input schema, and the graph contains information that applies to *any* application of the stylesheet to an input document that is valid with respect to the input schema.

The information derivable from the flow graph is essential in all subsequent analyzes in the XSLT2 tool kit: The validator assembles its summary graph to resemble the control flow graph; however, the summary graph is polyvariant, whereas the flow graph is monovariant. Dead flow and code analysis, and virtually any other static analysis of a stylesheet will depend on a good flow graph. The most important information derivable from a flow graph is:

- Given a template, the *context set* of declared node types that may be type of the context node of the template at some instance at runtime
- Given a *template-invoking instruction* (`apply-templates`, `apply-imports` or `next-match`), a context type and a template, the *edge flows* of declared node types that may be selected by the instruction when its context node has the context type, and flow to the template.

The very basic algorithm is taken over from previous work ([23]), but most of its superstructure have been redesigned, to satisfy new criteria:

- Aiming for a tool that can enhance an XSLT2 development process, the control flow algorithm must, besides computing flow, also be able to provide the user with information on what we will call *absent flows* — template recursions that do not materialize at run-time because the input schema does not permit the necessary structure for them to happen.
- XSLT2 has several new control flow-related constructs that also need to be accounted for: Mode values and `next-match`.
- Flow analysis is largely XPath analysis. The new features of XPath2 needed to be included.
- XML Schema is a larger language class that DTD, and some features of the old algorithm needed to be extended for that.

- It was found that the worst obstacle for successful application in an on-line editing context was time and memory performance. The work had its focus turned towards that, and resulted in a new algorithm that was much more satisfactory.

The overall contributions of this work to static XSLT flow analysis are:

1. An extension of the algorithm to report absent template recursions
2. An extension of the algorithm to work with XSLT2 and XPath2
3. An extension of the algorithm to work with XML Schema
4. A new flow algorithm, with significantly improved time and memory performance, compared to the previous algorithms. All the extensions to the algorithm were incorporated, and there was no loss of precision.

The description of the algorithm is rather detailed, making the result of the work practically implementation-ready.

Section 13.2 presents the problem to be solved in sufficient detail and Section 13.3 describes a general fixed point solution, allowing functions representing the different restrictions on flow that will exist in the combination of the stylesheet and the input schema to be "plugged" into it.

Section 13.4 presents a function modeling how template modes restrain flows, and Section 13.5 presents a function representing the semantics of the different template-invoking instructions. No XSLT flow analysis would be complete without these.

The rest of the functions all examine XPath path expressions to determine if a flow is feasible: Section 13.7 describes a simple, efficient but only moderately precise function, that will be put into service as an early eliminator of most infeasible flows. The remaining functions are more precise, but also computationally heavier: The abstract evaluation function of Section 13.8 and the ancestor language function of Section 13.9 are both candidates for the final stage of the analysis, where the few, hardest to rule out flows are found. It will be shown which different advantages and disadvantages the two have.

Section 13.11 describes how the functions can be composed into a more efficient flow analysis algorithm without loss of precision, and an experiment is performed, finding a much *filter* method for boosting its performance than what was previously used. More experimental results are presented, indicating a substantial improvement compared with previous work, but still with some things left to be desired.

Section 13.11 and Section 13.13 present two new algorithms invented in this work to decide certain aspects of flow feasibility in an apparently less general, but still simple, and much faster way than before. We will then prove that the new algorithms have at least as good precision as the composite algorithm they replace, and that there is no loss of generality.

A final, composite algorithm will then be presented in 13.12, and a series of practical test runs will show that its precision is the same as, or better than that of previous algorithms, but its application area is wider, and it runs much faster and in much less memory.

Finally, Section 13.15 concludes.

13.2 Definitions

Under pedantic correctness, the names of some of the functions defined below should be parameterized by the simplified stylesheet T, since they may change when T is changed. We omit that, and just refer to a given T where necessary.

Let $\mathcal{M}$ be the XSLT2 mode names; the set of valid **QName**s, plus the special token #**default**; a default, unnamed mode.

Let $\mathcal{R}$ be the template rules in T, and $\mathcal{I}$ be the template-invoking instructions in T. Let

$match : \mathcal{R} \rightarrow \mathbf{XPath}_{pattern}$ return the match pattern of each template, and
$select : \mathcal{I} \rightarrow \mathbf{XPath}_{path}$ return the selection expression of each template-invoking instruction: For an `apply-templates` instruction, that is its `select` expression; for the remaining template-invoking instructions, `self::node()` is used, enabling a uniform treatment of all template-invoking instructions while preserving semantics.

$r_i : \mathcal{I} \rightarrow \mathcal{R}$ returns the template rule containing a template-invoking instruction.

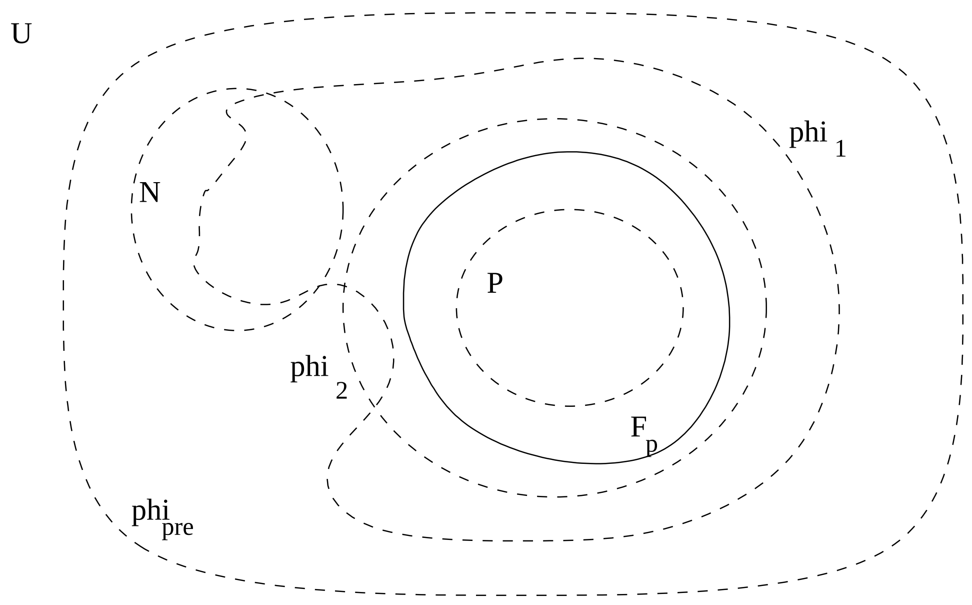
We base the analysis on the context model of [15], Section 5.4: At runtime, at any given time during the execution of a stylesheet, a certain template rule, the *current template rule* is in control, a certain XPath node is the *context node* (the official name is now context item, but it is known for sure that context items of templates are nodes), and a certain mode is the *current mode*. Statically, we abstract runtime instances to their types, and call the declared node type (see 6.4) of a context node its *context type*.

`xslv:unknownSequence()` `select` expressions in `apply-templates` instructions are just considered incompatible with everything in the analysis.

The *flow* described is the run-time transfer of *control* between template rules, that carries along with it a context node and a mode. The term *context flow* will also be used; this is the mapping of one context type to a set of context types, when a path expression is abstractly evaluated.

An *edge flow* with mode m and context type q' from a template-invoking instruction i to a template rule r' is a static abstraction that is may happen that a node of type q' flows from i to r' during a valid execution of T, invoking r' under mode m.

The goal is to construct a graph that abstractly describes, for any valid execution of S, the DNTs that may become context types for each template rule, and in turn which DNTs may



U (surrounding the figure) is a universe of all edge flows that can be described in terms of some given stylesheet and schema.

F_p is the set of edge flows that may exist during execution of the stylesheet, given that input documents are valid with respect to the input schema, also known as the exact edge flow set.

ϕ_{pre}, ϕ_1 **and** ϕ_2 are examples of upper approximations of F_p — containing all of it, and possibly more.

N is an example of a set of describable edge flows that a lower-approximating *negative* flow algorithm can guarantee will never exist during execution

P is a *lower* approximation of F_p.
Static analysis precision can be enhanced by:

- For a single flow, using as an upper approximation $\phi_1 \cap \phi_2$, instead of just either
- If, for some reason, ϕ_2 is not used, $\phi_1 \backslash N$ is better than ϕ_1 alone.

Static analysis performance can be enhanced by:

- First excluding most of U by running a complete analysis, using a fast test like ϕ_{pre}
- For a single flow, if the ϕ_1 test runs significantly faster than the ϕ_2 test, run the ϕ_1 test first, and then run the ϕ_2 test if the flow passed ϕ_1 (filtering)
- For a single flow, if the P test runs significantly faster than the most precise upper-approximate test, run that test only if the P test was negative (stuffing).

We used all of these observations in the design of the flow analysis algorithm.

flow from which template-invoking instructions to which template rules:

$G = (C, F)$

where

> Σ_d is the declared node types of the *input* schema D_{in}
>
> $C : \mathcal{M} \to \mathcal{R} \to 2^{\Sigma_d}$ describes the *context sets* for the template rules. These are upper approximations of the set of types of possible context nodes for a template $\in R$
>
> $F : \mathcal{M} \to \Sigma_d \to \mathcal{I} \to \mathcal{R} \to 2^{\Sigma_d}$ upper-approximates sets of edge flows from a template-invoking instruction $\in I$ to a template $\in R$, given a current mode $\in \mathcal{M}$ and a context type $\in \Sigma_d$

Conservativity of the whole validity analysis requires the graph to be "on the large side", yielding these requirements:

Conservativity of context sets: If there exists a document valid wrt. to d, such that at some time during application of T to the document, $m \in \mathcal{M}$ is the current mode, $q \in \Sigma_d$ is the current context type and $r \in \mathcal{R}$ is the current template, then $q \in C(m)(r)$.

Conservativity of edge flows: If there exists a document valid wrt. to d, such that at some time during application of T to the document, the current mode is $m \in \mathcal{M}$, the current context node has type $q \in \Sigma_d$ and the template-invoking instruction $i \in \mathcal{I}$ transfers control to the template rule $r' \in \mathcal{R}$, with a context node of type $q' \in \Sigma_d$, then $q' \in F(m)(q)(i)(r')$.

Integrity: The context set of every template rule must contain the context types of flows that flow into the template rule.

Nothing was said about anything required *not* to be included in C and F, so an obvious solution that satisfies the requirements is, of course, to include all DNTs in all values of C and F. This "solution" would hardly lead to anything that is able to provide the user with interesting information. As we shall see later, a "larger" graph uniformly transforms to a larger set of spurious errors — situations that will be reported as static errors but really are the effects of approximations in the analysis.

Two of our ultimate success criteria for an analysis tool in a wider setting return here:

- To maintain a pleasant user experience — the algorithm must have good time and memory performance
- To build user confidence — spurious errors should be kept to a minimum

The engineering challenges of this analysis are:

- Flow analysis should be efficient enough for on-line evaluation in an editor on a standard desktop computer
- The constructed control flow graph should be accurate enough to avoid most spurious errors. (one previously used success criterion was to have a generic identity transform validate OK with all schemas in a test set, plus evaluation of a corpus of practical use cases)

13.3 The fixed point flow algorithm

Traditionally [24], a fixed point algorithm starts out with a lattice, a control flow graph and a set of constraints, but in this case, a control flow graph is part of the result, not a prerequisite. The "control flow graph" used at start-up is a directed graph with a node for every template rule and every template-invoking instruction. It will have an edge from every template rule to its contained template-invoking instructions, and one from every template-invoking instruction to every template rule. The lattices used are (at largest) subsets of $\Sigma_d \times \mathcal{M}$.

Under the fixed point algorithm, these constraints will satisfy the requirements:

Initialization: The *initial* template rule as of the [15], Section 2.3, that is, the template rule r that is initially invoked with the initial context type q under the initial mode m, must have $q \in C(m)(r)$.

Flow propagation: When some node type q becomes a new context type of template rule r under mode m, for each template-invoking instruction $i : r_i(i) = r$ and for each template rule r', then if there exists a document valid wrt. d, such that a context flow at r maps a node of type q to a node of type q', and template r' is directly invoked with q' as its context node, $q' \in F(m)(q)(i)(r')$.

Integrity: This final constraint simply enforces that in-flowing types become context types: $F(m)(q)(i)(r) \subseteq C(m)(r)$

For now, we will solve the seemingly difficult flow propagation constraint simply by defining a class of Φ-*functions* fulfilling it, and then look for realizations of such functions.

Φ-functions

Define a Φ-function to be any function $\phi : \mathcal{M} \times \Sigma_d \times \mathcal{I} \times \mathcal{R} \rightarrow 2^{\Sigma_d}$, such that for the stylesheet S, for any input document valid wrt. to d, and for any combination of (m, q, i, r') of

- a mode m,
- a DNT q,

- a template-invoking instruction i in T
- and a template rule r' in T,

$\phi(m, q, i, r')$ is a superset of the types of the nodes that may be selected by i when m is the current mode, q is the context type, and $r_i(i)$ is the current template rule, and become context nodes of a flow from i directly to r'.

If ϕ is a Φ-function, then the constraint

$$q \in C(m)(r) \Rightarrow \phi(m, q, i, r') \subseteq F(m)(q)(i)(r')$$

where $r = r_i(i)$, will by the definition satisfy conservativity of edge flows locally, and we can rely on the fixed point algorithm, with suitable initialization, to satisfy the two other conservativity criteria.

Notice that Φ-functions have the property that the intersection $\bigcap_j \phi_j$ is a Φ-function if all of ϕ_j are.

In the following, we will present six Φ-functions:

- A *mode-compatibility* function was designed for the extended mode system of XSLT2, and is a new design.
- An *`apply-imports/next-match`* priority compatibility function is also a new design, accommodating for the template-invoking instructions that restrict the range of their target template rules. The `apply-imports` instruction was apparently ignored in previous work on static XSLT validation[23].
- A *schemaless* function; originally the idea of [11], and sharpened and extended for XSLT2 by us.
- An *abstract evaluation* function, which is a re-branding of previous work on static XSLT analysis[23], extended for XSLT2. Later on, it is extended and refined further, and its internals are put to new uses.
- An *ancestor language* function that was also derived from previous work ([23]), and extended for single-type schema languages.
- A new *fast flow function*, which is a primary result of this work, based on composing the above functions in a new way, and using new sub-algorithms.

After a detailed description of these functions, we are able to describe the special characteristics of each of them, and describe the experiments and results of composing them into an analysis algorithm.

13.4 A mode-compatibility Φ-function

XSLT2 template modes allow stylesheet authors to produce different transformation results for multiple transforms of the same input node by restricting flow, so a flow analysis can benefit from incorporating modes. Each template rule has a set of modes associated (from the `mode` attribute of the template rule) of incoming flow that the rule applies to, or instead, the special token #**all**, which means that the rule applies to flow of any mode. A mode is a member of $\mathcal{M}$.

When a template rule is invoked, a mode is passed along with the rest of the dynamic context, becoming the *current mode* for the template invocation. Each template-invoking instruction in the template rule in turn selects a new mode, and, if the instruction passes control to other template rules, it does so only to those rules that are applicable to the new mode. For `apply-templates`, the mode may be given in the stylesheet, or a special token #**current** may be used to pass the current mode. The other template-invoking instructions are hard-wired to #**current**. Our first Φ-function will model this. Let:

$$mode_r : \mathcal{R} \rightarrow 2^{\mathcal{M}} \cup \{\#\mathbf{any}\}$$

return the mode list for each template rule in the simplified stylesheet S, and let

$$mode_i : \mathcal{I} \rightarrow \mathcal{M} \cup \{\#\mathbf{current}\}$$

return the mode of each template-invoking instruction in S.

$$mode_i(i) = \begin{cases} m & \text{if } i = \texttt{<apply-templates mode="m" .../>} \\ \#\mathbf{current} & \text{otherwise} \end{cases}$$

The following functions reflect the semantics of template modes:

$app(M)$ returns a set of modes to which a template rule with mode list M is applicable:

$$\begin{aligned} app : 2^{\mathcal{M}} \cup \{\#\mathbf{all}\} &\rightarrow 2^{\mathcal{M}} \\ app(M) &= \begin{cases} \mathcal{M} & \text{if } M = \{\#\mathbf{all}\} \\ M & \text{otherwise} \end{cases} \end{aligned}$$

$modeflow(m, c)$ returns the mode of the flow out of some template-invoking instruction with mode m, given a current mode c. It simply states that #**current** passes on the current mode, and any other `mode` attribute specifies a new mode:

$$\begin{aligned} modeflow : (\mathcal{M} \cup \{\#\mathbf{current}\}) \times \mathcal{M} &\rightarrow \mathcal{M} \\ modeflow(m, c) &= \begin{cases} c & \text{if } m = \{\#\mathbf{current}\} \\ m & \text{otherwise} \end{cases} \end{aligned}$$

The mode-enforcing Φ-function now becomes:

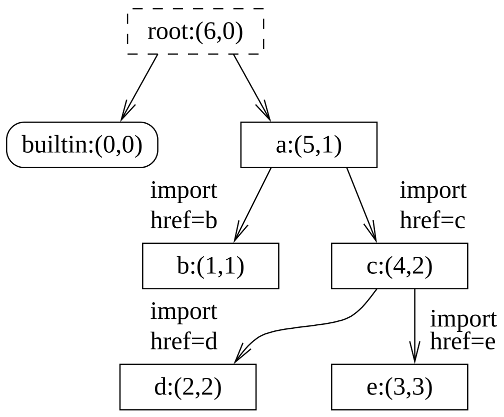


Figure 15: An import tree. The label of a stylesheet level l is formatted as: name of $l : (P(l), P_m(l))$. The built-in rules as a level is only conceptual.

$$\begin{aligned}
\omega_m(m,i,r) &= \begin{cases} \textbf{live} & \text{if } \mathit{modeflow}(\mathit{mode}_i(i),m) \in \mathit{app}(\mathit{mode}_r(r)) \\ \textbf{dead} & \text{otherwise} \end{cases} \\
\phi_m(m,q,i,r') &= \begin{cases} \Sigma_d & \text{if } \omega_m(m,i,r') = \textbf{live} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

It rules out that any flow ends up at template rules that are not applicable to the mode of the flow. This will not be the last time we will see modes, however: Later on, when introducing context-sensitive Φ-functions, we will make edge flows variant in modes.

A fixed point algorithm over this Φ-function will have subsets of $\mathcal{M}$ with and the subset relation as its lattice, causing a mode propagation to be queued for each time a mode flows to a template rule for the first time.

13.5 An `apply-imports/next-match` Φ-function

This Φ-function that models the semantics of the `apply-imports`/`next-match` instructions:

At the stylesheet load time, the *import tree*, as described in [15], section 3.10.3, is constructed. Then, the stylesheet levels are numbered in the order in which a post-order traversal of the tree will visit them:

$$P : \mathcal{L} \to \mathbb{N}$$

A lowest precedence of imported levels for each level is assigned:

$$\begin{aligned}
P_m &: \mathcal{L} \to \mathbb{N} \\
P_m(l) &= \min\{P(l)\} \cup \{P_m(l') | l' \text{ is imported from } l\}
\end{aligned}$$

Define $P_r : \mathcal{R} \to \mathbb{Q}$ as the priority of each template rule, and the *relative strength* of template rules as the composition of import precedence and priority:

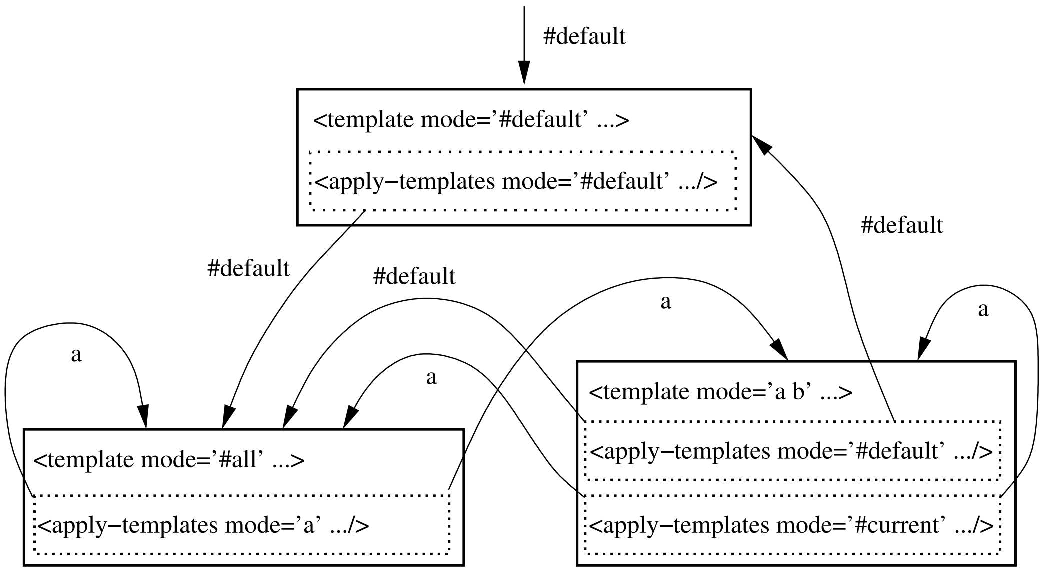

#default
<template mode='#default' ...>
<apply–templates mode='#default' .../>
#default
#default
#default
a
a
a
a
<template mode='a b' ...>
<apply–templates mode='#default' .../>
<apply–templates mode='#current' .../>
<template mode='#all' ...>
<apply–templates mode='a' .../>

Figure 16: An example of a mode flow graph.

$r_1 \geq r_2 \Leftrightarrow P(r_1) > P(r_2) \mid (P(r_1) = P(r_2) \ \wedge \ P_r(r_1) \geq P_r(r_2))$

Now, if $l_r(r)$ is the stylesheet level that contains the template r, and $l_{ri}(i) = l_r(r_i(i))$, we can pin down the largest set of template rules that may receive flow from a template-invoking instruction (as far as `apply-imports`/`next-match` is concerned):

$$v(i) = \begin{cases} \mathcal{R} & \text{if } i = \texttt{<apply-templates.../>} \\ \{r \in \mathcal{R}: \ \ P_m(l_{ri}(i)) \leq & \\ \qquad\qquad P(l_r(r)) < P_m(l_{ri}(i))\ \} & \text{if } i = \texttt{<apply-imports/>} \\ \{r \in \mathcal{R}: \ \ l_{ri}(i) \geq r \ \wedge \ r \not\geq l_{ri}(i)\} & \text{if } i = \texttt{<next-match/>} \end{cases}$$

The final Φ-function becomes:

$$\omega_a(m,a,r) \quad = \quad \begin{cases} \textbf{live} & \text{if } r \in v(a) \\ \textbf{dead} & \text{otherwise} \end{cases}$$
$$\phi_i(m,q,a,r') \quad = \quad \begin{cases} \Sigma_d & \text{if } \omega_a(a,m,r') = \textbf{live} \\ \emptyset & \text{otherwise} \end{cases}$$

A fixed point algorithm over this Φ-function will have as its lattice just $(\{\textbf{done}, \textbf{notdone}\}, \{\textbf{done} \sqsupseteq \textbf{notdone}\})$, limiting the number of visits to each template to 1.

13.6 Select-match compatibility

The following functions all determine *select-match compatibility* between template-invoking instructions and template rules by examining compatibility properties of `select` and `match` XPath expressions.

We will use the notation

$$u \overset{p}{\underset{x}{\longmapsto}} v$$

meaning: Given a XML tree fragment x, and a node u in the tree, evaluation of the XPath2 path expression p with u as the initial context node results in a sequence that contains a node v.

This notation will allow for simpler description of the tests that determine if edge flows exist from some template-invoking instruction to some template rule.

A necessary requirement for an edge flow from template-invoking instruction i to a template rule r'[40]

Necessary requirement for select-match compatibility:

There exists an XML tree fragment x containing the nodes n_1, n_2 and n_3, so

$$n_1 \overset{select(i)}{\underset{x}{\longmapsto}} n_2 \text{ and } n_3 \overset{match(r')}{\underset{x}{\longmapsto}} n_2$$

An abstraction of this to undeclared node types is the basis of the schemaless Φ-function in the next section. Another abstraction to declared node types is used in the abstract evaluation Φ-function in Section 13.8 and, with further refinements, in the ancestor language Φ-function in Section 13.9.

13.7 A Schemaless Φ-function

The next Φ-function in our flow analysis is capable of finding upper bounds for the values of the function C; the context sets for template rules, as well as for the values of the function F. The algorithm is a sharpened version of the *Raw-TAG* (Template and Association Graph) graph algorithm presented by [11]. The term "raw" refers to the algorithm *not* considering d — this is deferred to a later, *input schema-aware* Φ-function. The purpose of doing this is twofold: First, it will find a number of apparently very plausible flows that the schema-aware analysis will show not to exist after all — what Dong & Bailey call "invalid template calling relationships", and we term "absent flows": Control and context type flows that depend on structural features of the input tree which are not permitted by the input schema, and might surprise a user by their absence.

The second reason for wanting to perform this analysis — even if not interested in absent flows — is that it is a Φ-function that can be implemented very efficiently, so that when determining the absence of any flow at all for some combination of template-invoking instruction and template rule, this fact can be stored, and save us from coming to the same conclusion in a later stage of the analysis, when using a heavier Φ-function. In particular, the raw-TAG

[40]This models on the select-match semantics of XSLT1, and should be updated. For the present purpose, it does not harm, though.

Φ-function is *context insensitive* — it does not depend on its context type (q) parameter, which helps keep its associated lattice low.

This Φ-function uses a select-match compatibility test. The tests approximate towards too large context flows: A affirmative result for some certain context type, mode, template-invoking instruction and target template rule is not conclusive. A negative result is.

The analysis makes no use of input schema knowledge, but it is still accurate down to the level of element and attribute names.

Definitions

$\mathcal{Q}$ is the (infinite) set of valid namespace-qualified XML element or attribute names. The ***undeclared wildcard*** element type e_* represents any number of elements, each with with any name, and likewise, a_* represents any number of attributes, each with any name. These are needed as finite representations of unknown elements and attributes in a lattice and in an implementation. Σ_u is the set of all ***undeclared node types***, which are distinguishable in the XPath data model (no schema is used, so there are no declarations):

$$\begin{aligned}
\mathcal{E}_u &= \{e_*\} \cup \{e_q | q \in \mathcal{Q}\} \\
\mathcal{A}_u &= \{a_*\} \cup \{a_q | q \in \mathcal{Q}\} \\
\Sigma_u &= \mathcal{E}_u \cup \mathcal{A}_u \cup \{\mathbf{root}, \mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\}
\end{aligned}$$

Abstracting $\longmapsto$ to undeclared node types,

$$U \xrightarrow{p} V$$

means: Given a set of undeclared node types U , there exists an XML tree fragment x with a node n of some undeclared type $\in U$, $n \overset{p}{\underset{x}{\longmapsto}} v$ and $v \in V$.

Using the abstraction, and allowing the types of n_1 and n_3 of 13.6 to be any undeclared type (thus not restricting anything), a select-match (i, r') compatibility condition for an edge flow of type q' becomes:

$$\Sigma_u \overset{select(i)}{\longrightarrow} \{q'\} \text{ and } \Sigma_u \overset{match(r')}{\longrightarrow} \{q'\}$$

Domains

$$\begin{aligned}
&U : \mathbf{XPath}_{path} \rightarrow 2^{\Sigma_u} \\
&U^{step}_{a::t}, U^{axis}_{a}, U^{test_e}_{t}, U^{test_a}_{t} : 2^{\Sigma_u} \rightarrow 2^{\Sigma_u}
\end{aligned}$$

Path expressions

These functions simulate evaluation of an XPath path expression on a set of nodes: They iterate step by step, from the left to the right, and for each step, they simulate a context flow over the axis step, and a filtering of nodes by the node test of the step. For `intersect` expressions, a good approximation is possible, because nodes in a concrete evaluation intersection will certainly have a type in the intersected type set. There is no good approximation for `except`, though: It may still select nodes of a type in the result of abstractly evaluating the right-hand side.

$$\begin{array}{lll}
U^{step}_{a::t}(\Omega) & = & \begin{cases} U^{test_a}_t(U^{axis}_{attribute}(\Omega)) & \text{if } a = \texttt{attribute} \\ U^{test_e}_t(U^{axis}_a(\Omega)) & \text{otherwise} \end{cases} \\
U(\epsilon) & = & \{\mathbf{root}\} \\
U(a :: t) & = & U^{step}_{a::t}(\{e_*, a_*, \mathbf{root}, \mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\}) \\
U(P/a :: t) & = & U^{step}_{a::t}(U(P)) \\
U(p_1\,\texttt{intersect}\,p_2) & = & U(p_1) \cap U(p_2) \\
U(p_1\,\texttt{except}\,p_2) & = & U(p_1)
\end{array}$$

The function $U(p)$ computes an upper bound for $\{X \subseteq \Sigma_u | \Sigma_u \xmapsto{p} X\}$, although approximately; it starts from a finite representation of Σ_u, and recurses once for each step. The ϵ mapping maps leading `/`'s to the **root** node type.

Walking the axes

$$U^{axis}_{child}(\Omega) = \begin{cases} \{e_*, \mathbf{comment}, \mathbf{pi}\} & \text{if } \mathbf{root} \in \Omega \text{ and } \Omega \cap \mathcal{E}_u = \emptyset \\ \{e_*, \mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\} & \text{if } \Omega \cap \mathcal{E}_u \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$U^{axis}_{parent}(\Omega) = \begin{cases} \{e_*\} & \text{if} \quad \Omega \cap (\mathcal{E}_u \cup \{\mathbf{comment}, \mathbf{pi}\}) = \emptyset \\ & \qquad \wedge \\ & \qquad \Omega \cap (\mathcal{A}_u \cup \{\mathbf{text}\}) \neq \emptyset \\ \{e_*, \mathbf{root}\} & \text{if} \quad \Omega \cap (\mathcal{E}_u \cup \{\mathbf{comment}, \mathbf{pi}\}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$U^{axis}_{ancestor}(\Omega) = \begin{cases} \{e_*, \mathbf{root}\} & \text{if } \Omega \neq \{\mathbf{root}\} \wedge \Omega \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$U^{axis}_{descendant}(\Omega) = U^{axis}_{child}(\Omega)$$

$$U^{axis}_{preceding}(\Omega) = \\ \begin{cases} \{e_*, \mathbf{root}, \mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\} & \text{if } \Omega \cap (\mathcal{E}_u \cup \{\mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$U^{axis}_{following}(\Omega) = \\ \begin{cases} \{e_*, \mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\} & \text{if } \Omega \cap (\mathcal{E}_u \cup \{\mathbf{root}, \mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$U^{axis}_{preceding-sibling}(\Omega) = U^{axis}_{following-sibling}(\Omega) = \\ \begin{cases} \{e_*, \mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\} & \text{if } \Omega \cap (\mathcal{E}_u \cup \{\mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{array}{lcl} U^{axis}_{self} (\Omega) & = & \Omega \\ U^{axis}_{ancestor-or-self} (\Omega) & = & U^{axis}_{ancestor}(\Omega) \cup U^{axis}_{self}(\Omega) \\ U^{axis}_{descendant-or-self} (\Omega) & = & U^{axis}_{descendant}(\Omega) \cup U^{axis}_{self}(\Omega) \\ U^{axis}_{attribute} (\Omega) & = & \begin{cases} \{a_*\} & \text{if } \Omega \cap \mathcal{E}_u \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\ U^{axis}_{xslv} (\Omega) & = & \{e_*\} \end{array}$$

These functions rather straightforwardly simulate axis steps. U^{axis}_{child} / $U^{axis}_{descendant}$ returns everything that can be a child, if any node type in the argument set can have children. U^{axis}_{parent} returns "any element" for argument sets of text and attributes only, and adds the root node type if there are also elements, comments or processing instructions. Otherwise, the argument is empty or it is the singleton $\{\mathbf{root}\}$, which have no parents. The rest of the axis step functions should need no further explanation.

Node tests

Abstract node tests discard those types whose instances cannot pass the XPath node test. Therefore, the result set is always smaller than the argument set, or made more specific through the substitution of an undeclared wildcard for a concrete type.

$$\begin{array}{lcl} U^{test_a}_{q} (\Omega) & = & \begin{cases} \{a_q\} & \text{if } a_* \in \Omega \\ \{a_q\} & \text{if } a_q \in \Omega \\ \emptyset & \text{otherwise} \end{cases} \\ U^{test_a}_{*} (\Omega) & = & \Omega \cap \mathcal{A}_u \\ U^{test_e}_{q} (\Omega) & = & \begin{cases} \{e_q\} & \text{if } e_* \in \Omega \\ \{e_q\} & \text{if } e_q \in \Omega \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

$$
\begin{array}{lclclcl}
U^{test_e}_{*} & (\Omega) & = & \Omega \cap \mathcal{E}_u & & & \\
U^{test_e}_{node()} & (\Omega) & = & U^{test_a}_{node()} & (\Omega) & = & \Omega \\
U^{test_e}_{text()} & (\Omega) & = & U^{test_a}_{text()} & (\Omega) & = & \Omega \cap \{\mathbf{pcdata}\} \\
U^{test_e}_{comment()} & (\Omega) & = & U^{test_a}_{comment()} & (\Omega) & = & \Omega \cap \{\mathbf{comment}\} \\
U^{test_e}_{pi(...)} & (\Omega) & = & U^{test_a}_{pi(...)}(\Omega) & & = & \Omega \cap \{\mathbf{pi}\} \\
U^{test_a}_{attribute(q)} & (\Omega) & = & U^{test_a}_{attribute(q,t)} & (\Omega) & = & U^{test_a}_{q}(\Omega) \\
U^{test_a}_{attribute(*)} & (\Omega) & = & U^{test_a}_{attribute(*,t)} & (\Omega) & = & U^{test_a}_{*}(\Omega) \\
U^{test_e}_{attribute(q)} & (\Omega) & = & U^{test_e}_{attribute(q,t)} & (\Omega) & = & \emptyset \\
U^{test_a}_{element(q)} & (\Omega) & = & U^{test_a}_{element(q,t)} & (\Omega) & = & \emptyset \\
U^{test_e}_{element(q)} & (\Omega) & = & U^{test_e}_{element(q,t)} & (\Omega) & = & U^{test_e}_{q}(\Omega) \\
U^{test_e}_{element(*)} & (\Omega) & = & U^{test_e}_{element(*,t)} & (\Omega) & = & U^{test_e}_{*}(\Omega) \\
U^{test_a}_{schema-element(q)} & (\Omega) & = & U^{test_e}_{schema-attribute(q,.)} & (\Omega) & = & \emptyset \\
U^{test_a}_{schema-attribute(q)} & (\Omega) & = & U^{test_a}_{q} & (\Omega) & & \\
U^{test_e}_{schema-element(q)} & (\Omega) & = & U^{test_e}_{q} & (\Omega) & & \\
U^{test_a}_{document-node(.)} & (\Omega) & = & \emptyset & & & \\
U^{test_e}_{document-node(.)} & (\Omega) & = & \Omega \cap \{\mathbf{root}\} & & &
\end{array}
$$

The different naming of the $test_a$ and $test_e$ name tests serves to forward information to each name test from the axis step preceding it, whether the principal node kind was attribute or element. This influences the semantics of name tests. `attribute()`, `element()`, `schema-attribute()` and `schema-element()` are XPath2-specific node tests, which may examine the schema-declared types of elements and attributes, as well as element substitution groups (see...). Because the tests do not use schema information, there is no choice but to approximate to the larger: `element`() and `attribute`() ignore types and just match on names, and `schema-element`(q) matches on any element, since any element is potentially member of a substitution group headed by e_q. All signatures for `processing-instruction(...)` are represented as just `pi(...)` here. fix ref

The different overloads for the item tests `schema-element()`, `schema-attribute()` (typed or not, nillable or not) are all treated the same, and trivial repetitions have been omitted; the same goes for `document-node()` that may take an element test as a parameter.

The Φ-function

$$
\begin{array}{lcl}
\omega_u(m,i,r) & = & \begin{cases} \mathbf{live} & \text{if } U(select(i)) \hat{\cap} U(match(r)) \\ \mathbf{dead} & \text{otherwise} \end{cases} \\
\phi_l(m,q,i,r') & = & \begin{cases} \Sigma_d & \text{if } \omega_u(m,i,r') = \mathbf{live} \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

The $\hat{\cap}$ operator is a nonempty-intersection test that considers e_* and a_* semantics.

It might seem wasteful to reduce selections from Σ_u to emptiness or not, throwing away the type information. However, without making use of any schema knowledge at all, any information gained is readily lost: Most `select` expressions contain steps over other axes than `self`, which causes a loss of almost all information that could have been saved. Besides, abstaining from recycling (almost worthless) information lowers the lattice for the fixed point algorithm.

On the other hand, it may seem unnecessarily complex to evaluate every step of an expression, given that any information about elements selected by name etc. in all but the last steps are lost anyway. However, the Φ function is still slightly more precise this way while remaining simple conceptually, and , in the whole validation process, any performance cost of this will hardly matter. If desired, U in the ϕ_u expression could be replaced by U':

$$U'(P/a :: t) = U(a :: t) \ .$$

The results in this report were obtained with the original, all-steps variant of ϕ_u[41].

For the fixed point algorithm over this Φ-function, the lattice will be $\mathcal{M}$, with the subset relation as ordering. This will cause flow propagation each time a new mode reaches a template rule.

13.8 An abstract evaluation Φ-function

Like the schemaless function, this Φ-function is a select-match compatibility test, further taking into account a *context type* and the input schema of the analysis: The result of this test depends on the value of the context type parameter (q).

The idea of an abstract XPath evaluation select-match compatibility test originates from [23]. We have sharpened it, extended it for XPath2, and, as part of the extension of the general XSLT static analysis algorithm from local-type to single-type schema languages, altered it to work over *declared node types*, instead of over declared node *names* as earlier: Nodes with the same name, but declared differently, *are* distinguished.

Abstracting $\longmapsto$ to DNTs,

$$U \xrightarrow[d]{p} V$$

means: Given a schema d and two sets of DNTs U and V, there exists a node n of DNT $u \in U$ in an XML tree fragment x that is valid wrt. d, so that

$$n \xmapsto[x]{p} v \text{ and } v \in V$$

A sharpened version of the edge flow condition, now considering schema restrictions, is: There exists an XML tree fragment x which is valid wrt. to d, with a node n_1 of declared type q, a node n_2 of type q' and a node n_3, so that

[41] A few trial runs with the U' version were performed, showing it to be slightly less precise and slightly *slower* than the original version.

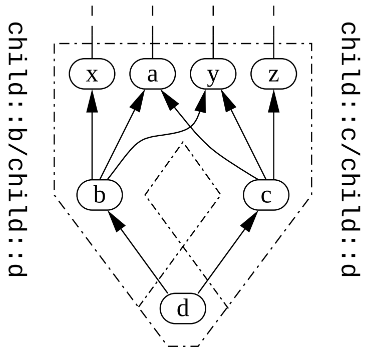


Figure 17: A pair of incompatible intermediate steps (the first steps of both expressions) that make the path expressions incompatible for the schema graphed.
The figure illustrates that if beginning abstract evaluation of each expression from context type a or y, the intermediate context sets after both first steps are disjoint but nonempty, making the steps incompatible. After a 2nd step, the two sets again have a nonempty intersection. Evaluating the paths separately, as in the abstract evaluation test, does not detect the incompatibility.

$$n_1 \overset{select(i)}{\underset{x}{\longmapsto}} n_2 \text{ and } n_3 \overset{match(r')}{\underset{x}{\longmapsto}} n_2$$

which implies:

$$\{q\} \overset{select(i)}{\underset{d}{\longrightarrow}} \{q'\} \text{ and } \Sigma_d \overset{match(r')}{\underset{d}{\longrightarrow}} \{q'\}$$

The function $S(p, \Omega)$ upper-bounds $\{X : \Omega \xrightarrow[d]{p} X\}$

Compatibility testing is composed from abstract evaluation of single XPath expressions simply as:

$$q' \in S(select(i), \{q\}) \cap S(match(r'), \Sigma_d) \ .$$

It should be obvious that DNTs not in the intersection of the select and match evaluation either are not possible results of the selection expression, or do not match the match pattern. However, because the test performs the two abstract evaluations independently, it tests the expression and the pattern only "end to end" — any effect that *incompatible intermediate steps* may have on compatibility is lost: Pairs of non-final steps of the two path expressions that have an empty intersection of accepted node types, and effectively make the expressions incompatible. Figure 17 is an illustration of one incompatible intermediate step going unnoticed. This loss led to [23]'s development of an even more precise test, described in the following section. The present test it still rather powerful, though: It has excellent time and memory complexity; it is conceptually simple, and precision is still quite good.

The abstract evaluation test works similarly to the one for the schemaless flow grapher, except that it considers the constraints of a schema, limiting the sets of types that each axis step generates, and each node test accepts. The general explanations for each group of functions are as for the schemaless abstract evaluation functions.

Domains

$$S : \mathbf{XPath}_{path} \times 2^{\Sigma_d} \rightarrow 2^{\Sigma_d}$$
$$S^{step}_{a::t}, S^{axis}_{a}, S^{test_e}_{t}, S^{test_a}_{t} : 2^{\Sigma_d} \rightarrow 2^{\Sigma_d}$$

Path expressions

$$\begin{array}{lll}
S^{step}_{a::t} & (\Omega) & = \left\{ \begin{array}{ll} S^{test_a}_{t}(S^{axis}_{attribute}(\Omega)) & \text{if } a = \texttt{attribute} \\ S^{test_e}_{t}(S^{axis}_{a}(\Omega)) & \text{otherwise} \end{array} \right. \\
S & (\epsilon, \Omega) & = \{\mathbf{root}\} \\
S & (a :: t, \Omega) & = S^{step}_{a::t}(\Omega) \\
S & (P/a :: t, \Omega) & = S^{step}_{a::t}(S(P, \Omega)) \\
S & (p_1\, \texttt{intersect}\, p_2) & = S(p_1) \cap S(p_2) \\
S & (p_1\, \texttt{except}\, p_2) & = S(p_1)
\end{array}$$

The function S is the recursive abstract XPath evaluation function. The ϵ mapping maps leading `/`'es to the **root** node type.

Walking the axes

$$\begin{array}{lll}
S^{axis}_{child} & (\Omega) & = \bigcup_{n \in C_d(\omega) \wedge \omega \in \Omega} S^*_d(n) \\
S^{axis}_{parent} & (\Omega) & = \{n | n \in P_d(\omega) \wedge \omega \in \Omega\} \\
S^{axis}_{ancestor} & (\Omega) & = \mathit{fix}(S^{axis}_{parent}(\Omega) \cup S^{axis}_{parent}(S^{axis}_{parent}(\Omega))) \\
S^{axis}_{descendant} & (\Omega) & = \mathit{fix}(S^{axis}_{child}(\Omega) \cup S^{axis}_{child}(S^{axis}_{child}(\Omega))) \\
S^{axis}_{preceding} & (\Omega) & = \left\{ \begin{array}{ll} \Sigma_d \backslash \mathcal{A}_d & \text{if } \Omega \neq \emptyset \text{ and } \Omega \neq \{\mathbf{root}\} \\ \emptyset & \text{otherwise} \end{array} \right. \\
S^{axis}_{following} & (\Omega) & = \left\{ \begin{array}{ll} \Sigma_d \backslash (\mathcal{A}_d \cup \{\mathbf{root}\}) & \text{if } \Omega \neq \emptyset \\ \emptyset & \text{otherwise} \end{array} \right. \\
S^{axis}_{preceding-sibling} & (\Omega) & = \\
S^{axis}_{following-sibling} & (\Omega) & = \left\{ \begin{array}{ll} \Sigma_d \backslash (A_d \cup \{\mathbf{root}\}) & \text{if } \Omega \backslash \mathcal{A}_d \neq \emptyset \\ \emptyset & \text{otherwise} \end{array} \right. \\
S^{axis}_{self} & (\Omega) & = \Omega \\
S^{axis}_{descendant-or-self} & (\Omega) & = S^{axis}_{descendant} \cup S^{axis}_{self} \\
S^{axis}_{ancestor-or-self} & (\Omega) & = S^{axis}_{ancestor} \cup S^{axis}_{self} \\
S^{axis}_{attribute} & (\Omega) & = \{n | n \in A_d(\omega)\ \wedge\ \omega \in \Omega\} \\
S^{axis}_{xslv} & (\Omega) & = \{\text{All element declarations made at stylesheet simplification}\}
\end{array}$$

The $\bigcup_{n \in C_d(\omega)\, \wedge\, \omega \in \Omega} S^*_d(n)$ expression on the `child` axis step is one of the "blessings" of XML Schema; it selects all declared substitutes of all element declarations (explicit or implied from derivation by extension) in Ω.

$\mathit{fix}(f(x))$ means fixed point; the function f is applied repeatedly to its value until $f^{n+1}(x) = f^n(x)$, which is the result. All of C_d, P_d and A_d and S^*_d were defined in Section 6.8.

Node tests

(Variable names are bound where mentioned on the left hand side, otherwise unbound)

$$
\begin{array}{llclcl}
S^{test_e}_{q} & (\Omega) & = & \Omega \cap \{e^i_q \in \mathcal{E}_d\} & & \\
S^{test_e}_{*} & (\Omega) & = & \Omega \cap \mathcal{E}_d & & \\
S^{test_a}_{q} & (\Omega) & = & \Omega \cap \{a^i_q \in \mathcal{A}_d\} & & \\
S^{test_a}_{*} & (\Omega) & = & \Omega \cap \mathcal{A}_d & & \\
S^{test_e}_{node()} & (\Omega) & = & S^{test_a}_{node()}(\Omega) & = & \Omega \\
S^{test_e}_{text()} & (\Omega) & = & S^{test_a}_{text()}(\Omega) & = & \Omega \cap \{\mathbf{pcdata}\} \\
S^{test_e}_{comment()} & (\Omega) & = & S^{test_a}_{comment()}(\Omega) & = & \Omega \cap \{\mathbf{comment}\} \\
S^{test_e}_{pi(...)} & (\Omega) & = & S^{test_a}_{pi(...)}(\Omega) & = & \Omega \cap \{\mathbf{pi}\}
\end{array}
$$

$$
\begin{array}{llcl}
S^{test_a}_{attribute(*)} & (\Omega) & = & S^{test_a}_{*}(\Omega) \\
S^{test_a}_{attribute(*,t)} & (\Omega) & = & \{a^i_q \in \Omega | T_d(t, a^i_q) = \mathbf{true}\} \\
S^{test_a}_{attribute(n)} & (\Omega) & = & S^{test_a}_{n}(\Omega) \\
S^{test_a}_{attribute(q,t)} & (\Omega) & = & \{a^i_q \in S^{test_a}_{q}(\Omega) | T_d(t, a^i_q) = \mathbf{true}\} \\
S^{test_e}_{schema-element(q)} & (\Omega) & = & \{e^j_{q'} \in S^*_d(e^i_q \in \Omega) | T_e(e^i_q, e^j_{q'}) = \mathbf{true}\}
\end{array}
$$

$$
\begin{array}{llcl}
S^{test_e}_{element(*)} & (\Omega) & = & S^{test_e}_{q}(\Omega) \\
S^{test_e}_{element(q)} & (\Omega) & = & S^{test_e}_{q}(\Omega) \\
S^{test_e}_{element(*,t?)} & (\Omega) & = & \{e^i_q \in \Omega | T_d(t, e^i_q) = \mathbf{true}\} \\
S^{test_e}_{element(q,t?)} & (\Omega) & = & \{e^i_q \in \Omega | T_d(t, e^i_q) = \mathbf{true}\} \\
S^{test_e}_{element(*,t)} & (\Omega) & = & S^{test_e}_{element(*,t?)} \cap \{q \in \Sigma_d | N_d(q) = \mathbf{false}\} \\
S^{test_e}_{element(q,t)} & (\Omega) & = & S^{test_e}_{element(q,t?)} \cap \{q \in \Sigma_d | N_d(q) = \mathbf{false}\}
\end{array}
$$

$$
\begin{array}{llclcl}
S^{test_a}_{element(q)} & (\Omega) & = & S^{test_a}_{element(q,t)}(\Omega) & = & \emptyset \\
S^{test_e}_{attribute(q)} & (\Omega) & = & S^{test_e}_{attribute(q,t)}(\Omega) & = & \emptyset \\
S^{test_a}_{schema-element(q)} & (\Omega) & = & S^{test_e}_{schema-attribute(q)}(\Omega) & = & \emptyset \\
S^{test_a}_{schema-attribute(q)} & (\Omega) & = & S^{test_a}_{q} & & \\
S^{test_e}_{DNTs(Q)} & (\Omega) & = & S^{test_a}_{DNTs(Q)}(\Omega) & = & \Omega \cap Q
\end{array}
$$

The Φ-function

$$\phi_a(m, q, i, r') \quad = \quad S(select(i), \{q\}) \cap S(match(r'), \Sigma_d)$$

The lattice for the associated fixed point algorithm is here considerably higher than in the previous algorithms: $(2^{Sigma_d \times \mathcal{M}}, \subseteq)$, again reflecting that the target of a flow depends on both its context mode and its context type.

Again, many functions defined in Section 6.8 are used: The type tests T_d and T_e that mimic *derives-from* of Section 2.5.4 of the XPath2 Recommendation ([12]), and N_d that can tell if a DNT is a pseudo-type for a nilled element type.

This function presents the best modeling of XPath2/XML Schema constructs, and its implementation is very efficient (and done in all but an afternoon from the above formulae), but it has the weakness of not detecting incompatible intermediate steps. The next (and last non-composite) function presented will improve on that.

13.9 A high-precision ancestor-language Φ-function

Ancestor language select-match compatibility testing is already a classic in static XSLT analysis ([23][20]), and the function presented here refines further upon it. As the abstract evaluation Φ-function, is was extended from working only over declared node *names* to working over declared node *types*, as a consequence of the upgrade from local to single-type schema languages, it is no longer possible to simply map names to types, as before. This was dealt with by introducing a rather performance-costly type search, that was later improved.

This project was originally conceived to just extend the traditional flow analysis, using this function composed with the mode, `apply-imports`/`next-match` and schemaless functions, and with some of the internal mechanics of the abstract evaluation function as helper components where necessary. To be able to better compare this approach with a new one that has superseded it[42], we will describe it in detail.

The fundamental idea of the function is, as we have seen in Section 6.6, that *ancestor languages in DTD and XML Schema are regular and mutually disjoint.* XPath2 path expressions that use only downward axes (those that map nodes to their descendant nodes and their attributes), or the `self` axis, can also be translated into regular expressions, and in such a way that the languages of these REs will have a nonempty intersection with the ancestor languages of the DNTs that match the path expressions. In particular, a select-match compatibility test can be made by translating the selection and the match XPath expressions to two regular languages, and then intersecting them to obtain a single language containing ancestor strings of nodes that are *both* selectable by the selection expression, *and* match the match pattern.

This test is superior to the abstract evaluation test, because incompatible intermediate steps *are* considered, and there is a further sharpening technique available for it, which we will call *path extension*: Getting more potential incompatible intermediate steps from match pattern of the template rule that *contains* the template-invoking instruction that is the source of the flow tested. With these qualities, this test is, as already demonstrated by [23], sufficiently precise for all cases seen in real-life XSLT except the few very most pathological.

[42]this function is now redundant, but still exists in the demonstrator application, and can easily be plugged in as the active flow algorithm.

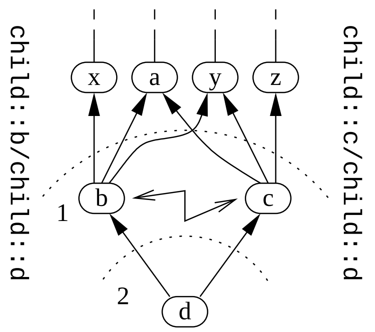


Figure 18: The ancestor language test detects an infeasible flow: $\mathcal{L}(\Sigma_d^*\texttt{bd}) \cap \mathcal{L}(\Sigma_d^*\texttt{cd}) = \emptyset$

13.10 Regular translation of path expressions

The desired regular expressions must have the conservativity property:

$$\{q\} \xrightarrow[d]{p} \{q'\} \Rightarrow anl_d(q') \cap \Sigma_d^* R(p) \neq \emptyset$$

Good precision is the converse:

$$anl_d(q') \cap \Sigma_d^* R(p) \neq \emptyset \Rightarrow q \xrightarrow[d]{p} q'$$

Define Γ_d to be the subset of Σ_d of node types that may appear as content:

$$\Gamma_d = \mathcal{E}_d \cup \{\mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\} \ .$$

In the regular expressions following, we will loosely use subsets and elements of Σ_d to represent the *names* of those subsets or elements, as not to clutter up the regular expressions with $(\mu(\cdots))$.

Rather obviously, non-downward steps are generally impossible to translate to regular languages over ancestor strings, as this would require some sort of negative-length language REs — upward steps truncate an ancestor string, sibling steps alter the last symbols, and before- or after steps have all sorts of effects. Fortunately, XSLT2 patterns are composed of only downward steps, and can all be translated.

Assume for now that all XPath2 path expression p uses only downward steps (a fix for non-downward steps will be made). A quite precise translation, including node test sharpening, fulfilling conservativity is:

$$R(p) = \begin{cases} R(p_1) + R(p_2) & \text{if } p \text{ has the form } p_1 \mid p_2 \\ \epsilon & \text{if } p \text{ has the form } \texttt{self} :: t \\ \epsilon & \text{if } p \text{ has the form } q\, \texttt{self} :: t \\ & \quad \text{and } R(q) = \epsilon \\ R(q) \,\cap\, \Sigma_d^*.R_{self}^{test}(t) & \text{if } p \text{ has the form } q\, \texttt{self} :: t \\ & \quad \text{and } R(q) \neq \epsilon \\ R(q).R^{axis}(a) \,\cap\, \Sigma_d^*.R_a^{test}(t) & \text{if } p \text{ has the form } q\, a :: t \\ & \quad \text{and } \texttt{self} \notin a \ \text{ and } R(q) \neq \epsilon \\ R^{axis}(a) \,\cap\, \Sigma_d^*.R_a^{test}(t) & \text{if } p \text{ has the form } a :: t \\ & \quad \text{and } \texttt{self} \notin a \\ R(q\, \texttt{descendant::}t) + R(q\, \texttt{self::}t) & \text{if } p \text{ has the form} \\ & q\, \texttt{descendant-or-self} :: t \\ \textbf{root} & \text{if } p \text{ has the form } \texttt{/} \end{cases}$$

where $\texttt{self} \in a$ means that a is `self` or `descendant-or-self`.

$$R_a^{test}(t) = \begin{cases} n & \text{if } a \neq \texttt{attribute} \text{ and } (t = n \text{ or } t = \texttt{element(n)} \\ & \text{or } t = \texttt{schema-element(n, .)}) \\ @.n & \text{if } a = \texttt{attribute} \text{ and } (t = n \text{ or } t = \texttt{attribute(n)} \\ & \text{or } t = \texttt{schema-attribute(n, .)}) \\ \mathcal{E}_d & \text{if } t = \texttt{*} \text{ and } a \neq \texttt{attribute} \\ \mathcal{A}_d & \text{if } t = \texttt{*} \text{ and } a = \texttt{attribute} \\ \Sigma_d & \text{if } t = \texttt{node()} \\ \textbf{root} & \text{if } t = \texttt{document-node()} \\ & \quad \text{or } t = \texttt{document-node(element(e))} \\ & \quad \text{or } t = \texttt{document-node(schema-element(e))} \\ \textbf{pcdata} & \text{if } t = \texttt{text()} \\ \textbf{comment} & \text{if } t = \texttt{comment()} \\ \textbf{pi} & \text{if } t = \texttt{pi()} \text{ or } t = \texttt{pi(t)} \text{ or } t = \texttt{pi('t')} \\ \{\sigma | \sigma = \mu(q) \wedge q \in Q\} & \text{if } t = \texttt{DNTs}(Q) \end{cases}$$

$$R^{axis}(a) = \begin{cases} \Gamma_d & \text{if } a = \texttt{child} \\ \mathcal{E}_d^*.\Gamma_d & \text{if } a = \texttt{descendant} \\ \mathcal{A}_d & \text{if } a = \texttt{attribute} \end{cases}$$

The function is generalized from that of [20] to accept expressions that begin in a `self` axis step. A few explanations:

The first step of a path expression p is translated to a regular expression representing the axis context flow of the step. The expression is then sharpened by restricting its suffix language to also be a sublanguage of the length-1 regular language from the R_a^{test} function, modeling that a node test throws away some nodes. This is repeated for the other steps, and the resulting languages are concatenated.

We prefix regular path translations by the language Σ_d^*, allowing any prefix of ancestor strings. This is done in following up on abstracting nodes to *types* and nothing else: *Context* information, such as ancestry, are thrown out of the model. Note that this does not ruin the effect of the **root** prefixing, as **root** only appears as the first symbol of ancestor strings anyway.

The `DNTs` node test is introduced in the next section.

Select-match compatibility of a candidate edge with a context flow from q to q' is still:

$$\{q\} \overset{select(i)}{\underset{d}{\longrightarrow}} \{q'\} \text{ and } \Sigma_d \overset{match(r')}{\underset{d}{\longrightarrow}} \{q'\}$$

translating to (the leading Σ_d^* is for permitting any ancestors):

$$anl_d(q') \cap \mathcal{L}(\Sigma_d^*(q\Sigma_d^*) \cap \mathcal{L}R(select(i))) \neq \emptyset$$

and

$$anl_d(q') \cap \mathcal{L}(\Sigma_d^* R(match(r'))) \neq \emptyset$$

In a world of downward-only selection expressions, this Φ-function would work:

$$\phi_d(m, q, i, r') \quad = \quad \{q' \in \Sigma_d \mid anl_d(q') \cap \mathcal{L}_{mcs} \neq \emptyset\}$$

where

$$\mathcal{L}_{mcs} = \mathcal{L}(\Sigma_d^* \cap R(select(i)) \cap \Sigma_d^* R(match(r')))$$

Because selection expressions are not generally downward-only, this variant is not useful. A repair will be made shortly.

Path extension

The above regular language test does consider incompatible intermediate steps; however, it fails to make use of the information that may be derived from the *source* template rule of a flow: The context node must have passed its pattern's test, and from this, more context information about its ancestry etc. may be derived[43].

A significant improvement can be made by *extending the selection path expression* by the match pattern of $r_i(i)$ and the context type, in principle:

$$select(i) \mapsto match(r_i(i))/\texttt{self} ::\mu(q)/select(i)$$

reflecting that nodes that are initial nodes for evaluation of the selection are context nodes of the template rule, and must have matched its match pattern. The `self` $::\mu(q)$ inserted sharpens any wildcards (`*` or `node()`) often seen in the last step of patterns, to accept only

[43] A context node has also passed all tests of predicates in the pattern. This information would be useful for further sharpening; for example, a predicate testing `xsi:type=`... could filter out all but one DNT that resulted from a type derived by extension in XML Schema. We leave this for the future.

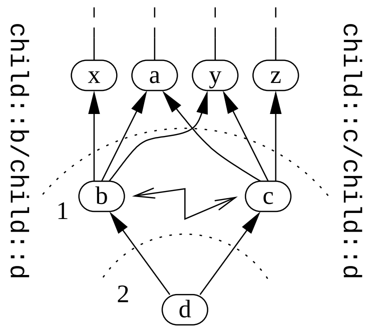


Figure 19: With path extension, the match pattern `child::b` of a template rule containing the selection expression `child::d` is prepended to the expression, enabling a regular ancestor language test to determine that no flow will go to a template rule with pattern `child::c/child::d`. Without path extension, the incompatibility would have gone unnoticed.

nodes with the same name as the context node, further refining the resulting extended path expression.

Selection path extension will also extend the length of the strings in the regular expression translation of the selection path, and, after Σ_d^* is prepended, the set of strings in the intersection of the selection language will be smaller:

Consider two regular expressions over Σ_d, r and $r_{ext}.r$. It should be obvious that any string $s \in \mathcal{L}(\Sigma_d^*.r_{ext}.r)$ is also in $\mathcal{L}(\Sigma_d^*.r)$, but the converse is not generally true: There may exist a string $s = x.y$ in $\mathcal{L}(\Sigma_d^*.r)$ no suffix of a prefix $x \in \mathcal{L}(r_{ext})$ exists for any suffix $y \in \mathcal{L}(r)$. $\square$

Extending a path p on the left by a step t will extend $R(p)$ (by $R(t) \cap \Sigma_d^* R_{\dots}^{test}(\cdots)$).

Path extension does not sacrifice conservativity; it merely applies contextual information to context types. It will provide an improvement in precision for all combinations of i and r', where the $select(i)$ has fewer steps than $match(r')$.

The above concatenation expression was an appetizer for the general idea, but it is not quite correct: Not all node types are converted to a proper test, and absolute selection paths will get the wrong semantics — they use the root node, not the context node, as their initial context. The proper version defines a combiner function c for match and selection expressions. If the selection expression is relative, it prepends to it a sharpened version of the match pattern:

$$\begin{array}{lcl} c & : & \mathbf{XPath}_{pattern} \times \mathbf{XPath}_{path} \times \Sigma_d \\ c(p_m, p_s, q) & = & \begin{cases} p_s & \text{if } p_s \text{ is absolute} \\ s_c(p_m, q)/p_s & \text{otherwise} \end{cases} \end{array}$$

The sharpener function s_c is:

$$s_c(p, q) = \begin{cases} /s_r(p_r, q) & \text{if } p = /p_r \text{ is absolute} \\ s_r(p, q) & \text{otherwise} \end{cases}$$

$s_r(p = p_1/\cdots/p_k, q) =$

$$\begin{cases}
p_1/\cdots/p_{k-1}/\texttt{attribute::}n & \text{if } k > 1 \text{ and } q = a_n^i \text{ and} \\
 & \begin{array}{lcl} (& p_k = & \texttt{attribute::*} \\ \text{or} & p_k = & \texttt{attribute::attribute()} \\ \text{or} & p_k = & \texttt{attribute::attribute(*)} \\ \text{or} & p_k = & \texttt{attribute::schema-attribute(*)} \\ \text{or} & p_k = & \texttt{attribute::schema-attribute(*,t)} \\ \text{or} & p_k = & \texttt{attribute::node()}\ ) \end{array} \\
\texttt{child::DNTs(}P\texttt{)/attribute::}n & \text{if } p = \texttt{attribute::*} \\
 & \text{and } q = a_n^i \text{ and } P_d(q) = P \\
\texttt{child::DNTs(}P\texttt{)/attribute::}n & \text{if } p = \texttt{attribute::node()} \\
 & \text{and } q = a_n^i \text{ and } P_d(q) = P \\
\texttt{child::DNTs(}P\texttt{)/attribute::}n & \text{if } p = \texttt{attribute::}n \\
 & \text{and } q = a_n^i \text{ and } P_d(q) = P \\
p_1/\cdots/p_{k-1}/a\texttt{::}m & \text{if } q = e_m^i \text{ and} \\
 & \begin{array}{lcl} (& p_k = & a\texttt{::*} \\ \text{or} & p_k = & \texttt{a::element()} \\ \text{or} & p_k = & \texttt{a::element(*)} \\ \text{or} & p_k = & \texttt{a::schema-element(*,t)} \\ \text{or} & p_k = & \texttt{a::schema-element(*,t?)}\ ) \end{array} \\
p_1/\cdots/p_{k-1}/a\texttt{::}m & \text{if } p_k = a\texttt{::node()} \text{ and } q = e_m^i \\
p_1/\cdots/p_{k-1}/a\texttt{::text()} & \text{if } p_k = a\texttt{::node()} \text{ and } q = \textbf{pcdata} \\
p_1/\cdots/p_{k-1}/a\texttt{::comment()} & \text{if } p_k = a\texttt{::node()} \text{ and } q = \textbf{comment} \\
p_1/\cdots/p_{k-1}/a\texttt{::pi()} & \text{if } p_k = a\texttt{::node()} \text{ and } q = \textbf{pi} \\
p & \text{otherwise}
\end{cases}$$

The multiple declaration test `DNTs`(Q) that accepts either of the DNTs in a set, is for internal analyzer uses and is not syntactically expressible in XPath2.

Repairing for non-downward steps

The test above can only handle downward and `self` axis steps, which is never a problem with patterns, but is not sufficient for selection expressions, which may be any path expressions. A fix for that is: Let

$l(p) =$ the lowest index i of a sequence so that $p_i/\cdots/p_{|p|}$ all have downward or `self` axes

and cut steps away from the expression, so only a rightmost path of downward and `self` steps, or ϵ, is left:

$$\begin{array}{lcl} w_p(p) & = & p_1/\cdots/p_{l(p)-1} \\ d_p(p) & = & p_{l(p)}/\cdots/p_{|p|} \end{array}$$

We could make do with using $\Sigma_d^* R(d_p(\mathit{select}(i)))$ as a regular language for the selection, but a sharpening using the context type q and $w_p(\mathit{select}(i))$ is possible:

$$
\begin{array}{lcl}
R_{any} : \mathbf{XPath}_{path} \times \Sigma_d & \rightarrow & \mathbf{Reg}(\Sigma_d) \\
R_{any}(p, q) & = & S(w_p(p), q)\Sigma_d^* \cap \Sigma_d^* R(d_p(p))
\end{array}
$$

(S was defined in Section 13.8) The final ancestor-language Φ-function becomes:

$$
\begin{array}{lcl}
R_s & : & \mathbf{XPath}_{pattern} \times \mathbf{XPath}_{path} \times \Sigma_d \rightarrow \mathbf{Reg}(\Sigma_d) \\
R_s(m, s, q) & = & \begin{cases} R(c(m, s, q)) & \text{if } l(s) = 1 \\ S(w_p(m)).\Sigma_d^* \cap \Sigma_d.R(w_d(m)) & \text{otherwise} \end{cases}
\end{array}
$$

$$\alpha = \Sigma_d^*.R_s(\mathit{match}(r_i(i)), \mathit{select}(i), q)$$

$$\beta = \Sigma_d^*.R(\mathit{match}(r'))$$

$$\phi_r(m, q, i, r') = \{q' \in \Sigma_d \mid \mathit{anl}_d(q') \cap \mathcal{L}(\alpha) \cap \mathcal{L}(\beta) \neq \emptyset\}$$

Solving the type search performance problem

ϕ_r, in its naïve form, has a serious performance problem: After $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$ is computed (as an automaton), the immediately apparent way to find the declared node types whose ancestor language intersects with $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$ is to test every declared node type's ancestor language automaton with $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$ for a nonempty intersection. A first experiment revealed that this test took up about 95% of the time of the whole ancestor language analysis — or, put differently, an already barely acceptable (for on-line use) algorithm had been made about 20 times slower.

[23] took advantage in his validation of DTD-specified XSLT that names and types were the same thing, and located all in-edges to accept states on $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$ automata — the declared node types sought after were then simply all types that had their name on such an edge, as a transition label. We are not as lucky, though, because more than one type may have the same name.

A solution was found: Using abstract evaluation for limiting the type search scope. It basically consists of running ϕ_a before ϕ_r, and performing type search in the ϕ_a result, which is, for all path expressions but those using the typed XPath2 `element()` and `schema-element()`[44] tests, a superset of the ϕ_r' result:

$$\phi_r'(m, q, i, r') = \{q' \in \phi_a(m, q, i, r') | \mathit{anl}_d(q') \cap \mathcal{L}(\alpha) \cap \mathcal{L}(\beta) \neq \emptyset\}$$

A further refinement is to use the edge-label technique after all, and then perform an nonempty-intersection test only on names that name more than a single declared node type. This was not tried.

[44]in which case precision is only improved

13.11 A composite fixed point algorithm

We now have Φ-functions that model modes, `apply-imports`/`next-match` semantics and find select-match compatible pairs with a quite descent precision, including recognition of some incompatible intermediate steps.

One composite Φ-function for the flow analysis (if adapted to use the same lattice) could be:

$$\phi_1 = \phi_m \cap \phi_i \cap \phi'_r$$

(That is, the mode function, the `apply-imports`/`next-match` function, and the ancestor language function — the schemaless and the abstract evaluation functions are not included, since they are generally less precise than the ancestor language function).

Obviously, there is no need to run all functions in all cases, and the computationally faster functions are better used for upper-bounding the results of the slower.

We instead split the fixed point algorithm into a sequence of three algorithms:

1. A fixed point algorithm with a lattice of subsets of $\mathcal{M}$, using $\phi_m \cap \phi_i$, and computing values for the function

   $F_m : \mathcal{I} \rightarrow \mathcal{M} \rightarrow 2^{\mathcal{R}}$

2. A fixed point algorithm with a lattice of subsets of $\mathcal{M}$, using ϕ_l as a constraint and F_m as its control flow graph, and computing values for

   $F_l : \mathcal{I} \rightarrow \mathcal{M} \rightarrow \mathcal{R} \rightarrow 2^{\Sigma_u}$

   F_l stores the values of $U(select(i)) \cap U(match(r'))$ to support dead flow reporting at a later stage.

3. A context-sensitive fixed point algorithm, with a lattice of subsets of $\Sigma_d \times \mathcal{M}$, using as its control flow graph:

   $$\begin{array}{lcl} F_u & : & \mathcal{I} \rightarrow \mathcal{M} \rightarrow 2^{\mathcal{R}} \\ F_u(i)(m) & = & \{r' | F_l(i)(m)(r') \neq \emptyset\} \end{array}$$

   — essentially the same as ϕ_l, but re-using F_l's data structure.

Looking back at Figure 13.2, this is the ϕ_{pre} performance enhancement introduced, with $\phi_{pre} = \phi_l \cap \phi_m \cap \phi_i$.

What exactly constituted the context-sensitive algorithm was the subject of some experiments. First, an experiment will demonstrate that the three-stage composition of the algorithms is soundly engineered.

Test case	l	t	e	a	n_1	n_2	n_3	n_4
adressebog	76	16	11	0	320	136	120	30
agenda	43	11	6	5	55	35	27	11
availablesupplies	42	10	18	5	40	24	10	2
dsd2html	1353	990	40	15	1200870	74387	8579	367
emaillist	257	54	20	1	345	460	162	43
links2html	128	20	12	1	340	340	340	185
ontopia2xtm	188	70	11	13	595	2074	434	45
order2fo	112	14	19	4	238	238	238	88
poem2xhtml	35	14	7	1	98	98	84	55
purchaseorder	112	18	22	6	306	114	96	31
slides2xhtml	118	21	13	2	231	210	281	70
sqlprocedures	258	20	14	3	300	300	280	50
staticanalysis	262	59	22	7	3599	1405	1453	195
window2xhtml	701	102	37	7	11832	2838	574	72
xhtml2fo	1697	462	89	119	235620	41319	11830	2430
xmlspec	2528	384	162	56	136704	32527	31351	22208

Table 4: Context-insensitive analysis of the [23] test corpus. For general information about the test environment, see Section 15.1.
l is the line count of the original stylesheet, t is the number of template rules in the *simplified* stylesheet, e is the number of element declarations in the test case input schema and a is the number of attribute declarations in the test case input schema.
n_1 is the number of context-less edges of (template-invoking instructions, mode, template rule) checked for mode and priority compatibility.
n_2 is the number of such triples surviving the (ω_m, ω_i) test, n_3 is the number of context-less edges tested by the schemaless algorithm and n_4 is the number surviving the ω_l test.
The large n_1 for the dsd2html test case is the result of 1213 template-invoking instructions being checked for mode compatibility with 990 template rules.

Test case	t_m	s_m	t_l	s_l	t	s	k_m	k_l
dsd2html	653	6.19	197	0.49	850	0.03	1.73M	375736
ontopia2xtm	6.0	34.9	8.4	18.9	14.4	7.56	646k	241548
staticanalysis	2.1	39.0	23.8	13.9	25.9	5.41	1.04M	50840
window2xhtml	7.0	24.0	9.8	2.54	16.8	0.61	1.28M	282245
xhtml2fo	161	17.5	326	5.88	487	1.03	1.21M	119291
xmlspec	82	23.8	1929	68.3	2011	16.2	1.27M	5349

Table 5: Evaluation of context-insensitive flow algorithm efficiency. t_m is the time for the mode and priority compatibility analysis, s_m is the test survival percentage, t_l is schemaless flow analysis time and s_l its survival percentage. t and s are total sums and products.
k_m and k_l are kill rates (edges/sec) for the two tests.
Test cases that analyzed in less than 10 ms are not shown. With the exception of the xmlspec case, the ϕ_m, ϕ_i and ϕ_l functions eliminated over 90% of the later flow search space, in less than 1 second.

Experiment: Performance of mode, `apply-imports/next-match` and schemaless analyzers

As can be seen in Table 4, the relatively simple mode, `apply-imports/next-match` and schemaless analyzers eliminate the vast majority of trivial cases, before the heavier, context-sensitive analyzers are set loose. They also complement each other very well, both eliminating a large fraction of infeasible flows (which is hardly surprising; the mode feature was designed to be complementary to other flow-related features).

It might be worthwhile to try invent more simple analyzers to add to the pipeline, or upgrade the schemaless analyzer to include undeclared node types in its lattice (it uses sets of unknown size, but the sets of element names that may be added *is* finite (limited by what is found in the stylesheet) — and a partial ordering, and least upper and greatest lower bound are all definable).

Conclusion: The combination of a mode, `apply-imports/next-match` and schemaless analyzers is appropriate for a front-end analysis. There are no performance problems in the schemaless analyzer, despite that it appeared complicated when written out formally.

A composite, context-sensitive Φ-function

The fixed point flow algorithm is based on the work-list algorithm, where the items in the work list are *candidate edges* — tuples of a mode, a declared node type, a template-invoking instruction and a target template rule, for testing with a Φ-function. Candidate edges are found using the control flow graph function F_u generated by the simple analyzers, with the f_m function of Section 13.4 and the already mentioned type search, for computing the mode and context type of an edge, given those of the originating template rule.

Instead of adding new candidate edges to the work list directly, performance can be enhanced considerably by *filtering* new-found candidates first, leaving some of the infeasible ones out of the work list. This effectively corresponds to using a *filter* function of more than one Φ-function, and evaluate one only if the other found a flow:

$$f(\phi_1, \phi_2)(m, q, i, r') = \begin{cases} \emptyset & \text{if } \phi_1(m, q, i, r') = \emptyset \\ \phi_2(m, q, i, r') & \text{otherwise} \end{cases}$$

As previous experience has shown, and as we will see again later, the $\phi_{r'}$ function is precise, but evaluates rather slowly, making the filter

$$f(\phi_a, \phi_{r'})$$

a candidate for a faster than $\phi_{r'}$ (in practice) function, but still at least as precise.

Experiment: Ancestor language vs. abstract evaluation filter

Both [23] and [20] propose a sort of filtering before context-sensitive analysis, with a context-insensitive ancestor language test.

It will not be described in detail here, because it is rather similar to the context-sensitive ancestor language test, with the exception that no context sharpening is done at path extension; the path extension that we presented as a refined version of:

$(select(i), q) \mapsto match(r_i(i))/\texttt{self}::\mu(q)/select(i)$

is simplified to:

$$select(i) \mapsto \begin{cases} match(r_i(i))/select(i) & \text{if } select(i) \text{ is relative} \\ select(i) & otherwise \end{cases}$$

Interestingly, [23] and [20] used the test differently: Olesen in a complete stand-alone fixed point analysis[45], similar to the way we use ϕ_l, and just with a **done/notdone** lattice[46], and Møller as a filter on the ancestor language analysis, caching automata for re-use. And interestingly, neither put very much faith in abstract evaluation: Both used it only as a last resort when everything else failed, for the non-downward step repair.

An experiment was performed, comparing the performance of a (context-insensitive) ancestor language filter function, and an abstract evaluation filter function. The experiment is interesting, because the two functions have different strong and weak points: The ancestor language function has the already described advantage of eliminating flows with incompatible intermediate steps, and its context insensitivity allows re-use of the same automata for all context types. The abstract evaluation function has the advantage of being context-sensitive, while still performing well.

The set-up for the experiment was: The mode, `apply-imports/next-match` and schemaless analyzes were run first to populate the data structures that implement the function F_u. The two different filters were set before ϕ'_r, as $f(., \phi'_r)$.

The conclusion immediately apparent from Table 7 is that a context-insensitive ancestor language filter tailgates the schemaless analyzer: In all of the larger examples, 93% or more of the flows that passed the schemaless test also passed the ancestor language filter. In general, the abstract evaluation filter scores much higher kill rates than the ancestor language filter, and it also kills infeasible candidate edges much faster. A noteable exception is the xhtml2fo test case, which has so many declarations in its schema that a context-insensitive filter won performance-wise (until beaten again by caching abstract evaluation results).

One reasonable explanation for a context-sensitive filter to perform much more precisely than a context-insensitive one is the high number of generic template-invoking instructions like ...`::*` and ...`::node()`. They are used very often, and the stylesheet simplifier adds

[45] At least, the implemented program of that project used it so.

[46] With XSLT1, there is no **#current** mode propagation to consider

Test case	n_1	n_2	n_3	n_4
dsd2html	4944	4938	4944	4763
ontopia2xtm	40	39	40	34
staticanalysis	166	155	166	95
window2xhtml	57	53	57	47
xhtml2fo	13748	13680	13860	3933
xmlspec	21393	20047	22573	2168

Table 6: Flow survival rates for the context-insensitive ancestor language vs. the abstract evaluation filters. n_1 is the number of flows tested by the ancestor language filter, n_2 is the number surviving. n_3 is the number of flows tested by the abstract evaluation filter and n_4 is the number surviving.

Test case	t_r	s_r	t_l	t_{lc}	s_l	k_r	k_l	k_{lc}
dsd2html	20364	99.9	1445	794	96.3	0.29	125	171
ontopia2xtm	38.8	97.5	3.8	2.8	85.0	25.8	1579	2142
staticanalysis	71.4	93.3	11.0	7.6	57.2	154	6455	9342
window2xhtml	87.6	93.0	6.4	6.4	82.5	45.7	1563	1563
xhtml2fo	13765	99.5	108942	31228	28.4	97.78	91.1	317
xmlspec	692072	93.7	142754	10793	9.60	1.94	144	1891

Table 7: Performance of the context-insensitive ancestor language vs. the abstract evaluation filters. t_r is the time in ms used by the ancestor language filter; s_r is its survival percentage. t_l is the time used by the abstract evaluation filter, and t_{lc} is the time used by the cached abstract evaluation filter; s_l is its survival percentage. k_r and k_l are kills per second of execution time for the three.
In all cases except xhtml2fo without caching, the abstract evaluation filter was superior.

more. For each of these, a context-insensitive filter will approve any template rule matching an element type, or any type at all, respectively, unless the template rule containing the template-invoking instruction has a sharp pattern. This property is largely shared with the schemaless flow analyzer, which was run before the analysis using filtering. On the other hand, a context-sensitive filter uses context information instead of any (generic) containing match pattern, and, with the simple implementation of abstract evaluation, it needs not be expensive.

Adding a cache that maps each template rule to an upper-approximated set of DNTs matching the pattern of the rule, and another cache

$$\mathbf{XPath}_{path} \rightarrow \mathit{Strings} \rightarrow \Sigma_d \rightarrow 2^{\Sigma_d}$$

storing evaluations of selection expressions from particular context DNTs, helped speed up the abstract evaluation filter. The cache values are reused for all selection expressions that are textually the same with predicates stripped off.

Conclusion from the experiment: At least while in the slipstream of the schemaless analyzer, the context-sensitive abstract evaluation filter performs better than the context-insensitive ancestor language filter. In the previous experiment, we saw that there is no reason not to use the schemaless analyzer, so for the later experiments, the ancestor language filter was abandoned in favor of the abstract evaluation filter.

Experiment: Overall flow algorithm performance, abstract evaluation filter

The previous two experiments focused on the environment of the context-sensitive "final" Φ-function, but we have not yet looked into the performance of that function proper. With the decision to use the abstract evaluation Φ-function as a filter, that leaves only the context-sensitive ancestor language Φ-function for the task. That is no coincidence, of course, as this is the most precise but also the slowest Φ-function.

A test run was performed, still with complete mode, `apply-imports`/`next-match` and schemaless analyzes performed first to populate the data structures that implement the function F_u, followed by a fixed point analysis using the Φ-function $\phi_x = f(\phi'_r)$.

As can be seen, performance was quite good in all cases, except the large xmlspec and xhtml2fo cases. It is also evident that the context-sensitive ancestor path test has a *very* low kill rate, leading us to think of ways to circumvent or completely eliminate automaton testing.

Simply removing the test and accepting all flows, as found by the context-sensitive abstract evaluation filter is not satisfactory, for two reasons: Incompatible intermediate steps would be regarded as compatible, which is hardly acceptable from a usability viewpoint, and there was not a lower-approximate test available for the *competition analysis* to be presented.

Test case	a	k	k/t_1	k/t_2	k/t_3
dsd2html	17764	0	0	0	0
staticanalysis	96	0	0	0	0
xhtml2fo	?	?	0	0	0
xmlspec	3383	160	0.82	1.44	1.46
links2html	62	3	56	-	-
sqlprocedures	30	9	196	-	-
all others		0	0	0	0

Table 8: Infeasible flow kills for ancestor language context-sensitive flow analyzer, after abstract evaluation filtering, for selected test cases. a is the number of edge flows found, k is the number of edge flows surviving the filter, but killed by the ancestor language algorithm.
k/t_1 is the kill rate of the uncached algorithm, k/t_2 is the kill rate of the α-caching algorithm and k/t_3 is the kill rate of the α and result caching algorithm

Test case	t_α	t_β	$t_{\alpha\beta}$	t_s	t_t	t_{ns}	m_{max}
dsd2html	13150	249	9337	44342	69014	24672	27M
ontopia2xtm	31	15	4	42	89	47	-
staticanalysis	132	21	19	54	193	139	3.3M
window2xhtml	71	29	7	84	221	137	-
xhtml2fo	?	?	?	?	"∞"	?	-
xmlspec	24289	2073	10252	132687	195701	63014	202M
links2html	20	9	4	15	54	39	
sqlprocedures	15	12	3	6	46	40	

Table 9: Performance of ancestor language context-sensitive flow analyzer, after abstract evaluation filtering, and with the result of the abstract evaluation as a type search scope limit.
t_α is the time used creating α-automata, t_β is the time used creating β-automata (they are cached at their respective templates) and $t_{\alpha\beta}$ is the time used intersecting α- and β automata.
t_s is the time used performing type search, t_t is the total time used by the ancestor language tests and $t_{ns} = t_t - t_s$ is a lower time bound on the ancestor language test; the time it would still take even if a zero-time type search was invented.
m_{max} was the maximum heap allocated, as seen from the GC trace.
The very large xhtml2fo test case had to be given up; it did not complete even after several hours.

Test case	t_α	t_β	$t_{\alpha\beta}$	t_s	t_t	t_{ns}	m_{max}
dsd2html	467	246	9277	44171	55917	11746	29M
staticanalysis	43	22	24	42	141	99	3.4M
xmlspec	23168	1634	6766	58943	110393	51450	261M

Table 10: Caching α-automata. All table column headers have the same meaning as in Table 13.11. α caching had little effect on the small staticanalysis case; the medium size dsd2html case remained dominated by the search time, while remarkably little happened with the larger xmlspec: Few XPath expression combinations mapped to the same cache keys. The xhtml2fo still did not terminate within the limits of our patience.

Test case	t_α	t_β	$t_{\alpha\beta}$	t_s	t_t	t_{ns}	t_{mm}
dsd2html	532	265	324	4477	7329	2852	6946
staticanalysis	44	20	13	37	127	90	691
xmlspec	22727	1648	6675	58437	109309	50872	150057

Table 11: Caching α-automata and the results of type search (keyed by a string hash derived from the XPath expressions that α and β are computed from, and the context type). The small staticanalysis2html case still saw little change (and it was adequately fast already); the medium dsd2html case now computed in 7 seconds, which is comparable with the time achieved by [23] in the *tmm* column (leading us to conjecture that he also cached). The larger xmlspec case still suffered from having overloaded its host computer.

An incompatible-paths test

It was found that *all* edge kills scored by the context-sensitive ancestor path automaton test were on the form: Selection expression `child::x/child::z` (possibly composed in path extension) incompatible with pattern `child::y/child::z`.

Recalling the requirement for edge flow in Section 13.6, a characterizing feature of the problem is that patterns conceptually read right-to-left starting from some *unknown* node, while expressions read left-to-right, and start from a context node; the evaluation of them all boils down to the final nodes selected and matched, so the two expressions should really be aligned "last step to last step" when inspected for compatibility.

Indeed, if restricting to `child` and `attribute` axes, and numbering steps backwards:

$$\begin{aligned} S &= a_n^s :: t_n^s \cdots a_1^s :: t_1^s \\ M &= a_m^m :: t_m^m \cdots a_1^m :: t_1^m \end{aligned}$$

it becomes apparent that, for some node:

- If the node fails *either* the node test t_1^s *or* the node test t_1^m, then it is either not selected, or not matched, so S and M are incompatible for the node.
- If $a_1^s \neq a_1^m$, then one axis step selected only content, and the other only attributes. S and M are incompatible (for any node at all).
- Both steps can only select nodes that are children or attributes of parents. If $a_n^s :: t_n^s \cdots a_2^s :: t_2^s$ is incompatible with $a_m^m :: t_m^m \cdots a_2^m :: t_2^m$ for *any* declared parent of our node, then and S and M are incompatible.

The argument goes on inductively, until the mth or the nth step, whichever is less, at which point there are no more opportunities to conclude incompatibility.

As an algorithm, and evaluating for a set of potential result types instead of just one:

Algorithm *incompatiblePaths*(S, M, Ω):

```
$i \leftarrow 1$
$\Omega_s \leftarrow \Omega$ ; $\Omega_m \leftarrow \Omega$
**while** $i \leq \min m, n$ **do**
  **if** $a_i^s \neq \texttt{child} \mid a_i^s \neq \texttt{attribute}$ **then**
    return **unsure**
  **end if**
  **if** $a_i^m \neq \texttt{child} \mid a_i^m \neq \texttt{attribute}$ **then**
    return **unsure**
  **end if**
  **if** $a_i^s \neq a_i^m$ **then**
    return **true**
  **end if**
  $\Omega_s \leftarrow S_{t_i^s}^{test}(\Omega_s)$ ; $\Omega_m \leftarrow S_{t_i^m}^{test}(\Omega_m)$
  **if** $\Omega_s \cap \Omega_m = \emptyset$ **then**
    return **true**
  **end if**
  $\Omega_s \leftarrow R_{a_i^s}^{axis}(\Omega_s)$; $\Omega_m \leftarrow R_{a_i^m}^{axis}(\Omega_m)$
  $i \leftarrow i + 1$
**end while**
return **unsure**
```

The R axis step functions are the S axis steps reversed:

$$\begin{array}{lcl} R_{child}^{axis}(\Omega) & = & \{n \in \Sigma_d | \omega \in C(n) \wedge \omega \in \Omega\} \\ R_{attribute}^{axis}(\Omega) & = & \{n \in \Sigma_d | n \in A_d(\omega) \wedge \omega \in \Omega\} \end{array}$$

Consider how the context-sensitive ancestor language test compares with this test for `child` and `attribute` axes only: The ancestor language test will test if

$$anl_d(q') \cap \mathcal{L}(\Sigma_d^* R_{a_m^m}^{test}(t_m^m) \cdots R_{a_1^m}^{test}(t_1^m)) \cap \mathcal{L}(\Sigma_d^* R_{a_n^p}^{test}(t_n^p) \cdots R_{a_1^n}^{test}(t_1^n)) = \emptyset$$

(all the $R_{\cdots}^{test}(\cdots)$ are length-1 languages). In other words, length-($\min(m, n)$) suffixes are maybe incompatible in the ancestor languages; prefixes are not.

Also consider how an NFA is constructed for such a `child`/`attribute` axis-mix path expression: Linearly. It has one state more than the length of the expression, and transitions are labeled with element or attribute names. Consider what happens when reversing such two automata, and intersecting them: That becomes identical to *incompatiblePaths*. So, for the simple case of `child`/`attribute` expressions of equal length, the two compatibility tests are equally precise.

That leaves automata with only `self`, `descendant` and `descendant-or-self` steps as cases to argue about where they may still *seem* sharper than *incompatiblePaths*, but

- h`descendant::`tr is equivalent to h`descendant-or-self::node()/child::`tr
- h`descendant-or-self::`tr is equivalent to h`descendant::`tr `|` h`self::`tr (but gives

us one more case to test)

for some prefix path expression h, suffix path expression r, axis a and node test t. Considering regular translation, we can also do:

- ha`::`t_1`/self::`tr_2r is equivalent to ha`::`$(t_1 \cap t_2)r$
- leading `self::`tr have the same translation as r and can be stripped.

The $(t_1 \cap t_2)$ "node test" is an intersection node test — the result is a standard XPath node test, or a "`false()`" test (use an absurd name test, then) for `*`, all name tests and `comment()`, `processing-instruction()` and `text()`; the more advanced XPath2 item tests approximate to name tests in regular translation, and can be treated as such.

— so for considerations about their regular translation, we might as well consider all path expressions as composed of just `child::`t, `attribute::`t and `descendant-or-self::node()` steps. Furthermore, arguing about select-match compatibility between two such expressions; a selection and a pattern, we are free to switch around the two at will (neither the regular test or *incompatiblePaths* make use of any special pattern properties), and we can remove any leading `/descendant-or-self::node()/` step from the pattern; it means nothing[47].

Assume that some DNT q' is both selectable and matchable:

$anl_d(q') \cap \mathcal{L}(\Sigma_d^* R(S)) \neq \emptyset$ and $anl_d(q') \cap \mathcal{L}(\Sigma_d^* R(M)) \neq \emptyset$

Assume that the *incompatiblePaths* test failed to detect incompatibility:

$\mathit{incompatiblePaths}(S, M, \{q'\}) =$ **unsure**

and also that the regular incompatibility test was sharper, and detected incompatibility:

$anl_d(q') \cap \mathcal{L}(\Sigma_d^* R(S)) \cap \mathcal{L}(\Sigma_d^* R(M)) = \emptyset$.

Let k be the largest k' such that the suffix expressions $S_{k'} \cdots S_1$ and $M_{k'} \cdots M_1$ have only `child` or `attribute` axes. Their regular translations are:

$R(S) = R(S_n \cdots S_{k+1}).R(S_k \cdots S_1)$ and $R(M) = R(M_m \cdots S_{k+1}).R(M_k \cdots M_1)$

There is a string $a \in anl_d(q') : a \in \mathcal{L}(\Sigma_d^* R(S))$, and a has a suffix

$a_k \cdots a_1 \in R(S_k \cdots S_1) \cap R(M_k \cdots M_1)$ (the NFA argument)

Then, is must be the case that $a_{|a|} \cdots a_{k+1} \notin \mathcal{L}(\Sigma_d^* R(S_n \cdots S_{k+1})) \cap \mathcal{L}(\Sigma_d^* R(M_k \cdots M_1))$.

and also one of:

[47]Although surprisingly many stylesheet authors do not realize that. The `//` *does* affect the default priority, though!

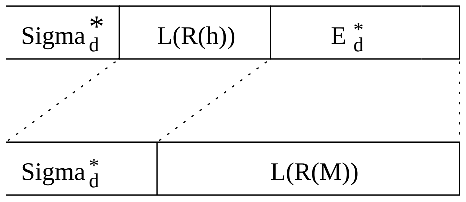


Figure 20: Illustration of $\Sigma_d^* R(h)\mathcal{E}_d^* x \cap \Sigma_d^* R(M)$

- Either $S_n \cdots S_{k+1} = \epsilon$ or $M_m \cdots M_{k+1} = \epsilon$

  One expression, say M, has en empty prefix; without affecting anything, we use `self::node()` as a prefix instead, and it translates to $\Sigma_d^* R(M_m \cdots M_{k+1}) = \Sigma_d^* R(\texttt{self::node()}) = \Sigma_d^*$

  The prefix implication becomes

  $a_{|a|} \cdots a_{k+1} \notin \mathcal{L}(\Sigma_d^* R(S_n \cdots S_{k+1})) \cap \mathcal{L}(\Sigma_d^*)$

  which contradicts $a \in \mathcal{L}(\Sigma_d^* R(S))$

- or (say) $M_m \cdots M_{k+1}$ is on the form h`descendant-or-self::node()`, where h has at least one step

  $R(M_m \cdots M_{k+1}) = R(h\texttt{descendant} :: \texttt{node}()) + R(h\texttt{self} :: \texttt{node}()) = R(h)\mathcal{E}_d^* \Sigma_d + R(h) \cap \Sigma_d^* \Sigma_d$

  Intersecting with $\Sigma_d^* R(M_m \cdots M_{k+1})$ has no effect on $\Sigma_d^* R(S_s \cdots S_{k+1})$ at all; see Figure 20 for an illustration, and we assumed that $a \in \mathcal{L}(\Sigma_d^* R(S))$.

  $anl_d(q') \cap \mathcal{L}(\Sigma_d^* R(S)) \cap \mathcal{L}(\Sigma_d^* R(M)) \neq \emptyset$

  again contradicting the assumption about the ancestor language test being sharper than *incompatiblePaths*.

Really, we still need to show that

$$q' \in S(S, \{q\}) \cap q' \in S(M, \{\Sigma_d^*\}) \ \Rightarrow \begin{cases} anl_d(q') \cap \mathcal{L}(\Sigma_d^* R(S)) \neq \emptyset \\ anl_d(q') \cap \mathcal{L}(\Sigma_d^* R(M)) \neq \emptyset \end{cases}$$

holds on the simplified expressions, so that abstract evaluation can replace automata entirely, but that should be rather obvious. Also, for absolute path expressions (we argued only for relative expressions above...), *incompatiblePaths* can be made as sharp as the regular language test by having it return **true** if it reaches the left-hand end of an absolute expression, if the other expression is longer.

What is the implication of this `descendant-or-self` weakness for the precision of the tool? Well, for a pattern $h//r$ the author of the stylesheet must have wanted to select only nodes that have an ancestry passing the h test, and themselves pass the r test.

The static abstraction, however, accepts the type of any node that passes the r test, and has a *declared* ancestor that passes the h test: The only case we can come up with, where the *incompatiblePaths* algorithm was **unsure**, but the ancestor language test could eliminate a flow of DNT q', is when S does not even select q', or M does not even match q'; that case will be caught in the abstract evaluation filter anyway.

The conclusion to all this is that there is no loss of precision in eliminating ancestor languages entirely from flow analysis; the simpler abstract evaluation and *incompatiblePaths* tests have the same power. This was further corroborated empirically: A test run resulted in *identical* (extremely verbose) flow graph dumps for all test cases using the ancestor language test vs. using the *incompatiblePaths* test.

We believe we are the first to present this kind of XPath path incompatibility test.

```
<!-- r1 -->
<template match="ingredients">
  <h:ul>
    <apply-templates select="list"/>
  </h:ul>
</template>

<!-- r2 -->
<template match="nutrition">
  <h:table>
    <apply-templates select="list"/>
  </h:table>
</template>

<!-- r3 -->
<template match="*">
  <apply-templates select="node()"/>
</template>

<!-- r4 -->
<template match="ingredients/list/item">
  <h:li><value-of select="name"/></h:li>
</template>

<!-- r5 -->
<template match="nutrition//item">
  <h:tr>
    <h:td><value-of select="name"/></h:td>
    <apply-templates/>
  </h:tr>
</template>
```

An example of a construct that will cause a spurious validation error: Suppose the input language has the declared element types `ingredients` and `nutrition`, both with a declared `list` child, which in turn has a declared sequence of `item` children. Then, r_1 might pass `list` elements with `ingredients` parents to r_3. Likewise, r_2 might pass `list` elements with `nutrition` parents to r_3. At runtime, the parent element distinction between the `list` elements flowing to r_3 is maintained; however, in our static analysis it is lost, *causing an apparent flow from* r_1 *via* r_3 *and on to* r_5 to appear, along with an apparent flow from r_2 via r_3 to r_4. The result is a complaint from XHTML validation that the child type `h:tr` is illegal for `h:ul`, and `h:li` for `h:table`, even though this mix-up will never happen at runtime. As usual, this kind of spurious errors may be confusing to non-insiders.

A possible improvement of this requires attaching more information to context types of edge flows, and using that information for sharpening select-match compatibility; in other words, heightening the lattice of the fixed point analysis.

The sharpest lattice candidate we imagined was one of *sublanguages of ancestor languages*, represented as regular languages and ordered by language inclusion. Such a lattice would cause the context types of the flow from r_1 to r_3 to be distinct from those from r_2 to r_3 (the former's ancestor sublanguage certainly has a `ingredients` as the 2nd last symbol in its strings; the latter's strings have a `nutrition`), and the two different types would in turn propagate different edge flows out of r_3, eliminating the spurious error. But such a lattice also has infinite height, causing a possible nontermination of the fixed point analysis (just consider a DNT declared as a descendant of itself, with a matching set of recursive template rules). Lattice *widening* could solve this; unfortunately, no efficient widening operation exists for regular languages, so the approach will not work.

Another candidate is a subset lattice over limited-length ancestor language suffixes, such that if using a suffix length limit 3 or more in our example, `item` nodes represented by `ingredient.list.item` become distinguishable from `nutrition.list.item` nodes. Such a lattice would solve half of our precision problem, making it possible to exclude flows of `nutrition.list.item` to r_4, but still not of the `ingredients`-ancestored `items` to t_5, for reasons explained in ... .

Fortunately, the kind constructs provoking the spurious errors described here seems to occur too rarely to justify the increased complexity of such "super" lattices, and there were no additional problems found in the transition from local-type to single-type languages in this respect. If the missing widening operation is invented, or if it is experienced that spurious errors of this kind confuses users of a tool based on this algorithm, one of the lattice heightening techniques could be added.

13.12 Fast flow algorithm

First, the extended path expression p

$$p = c(match(r_i(i)), select(i), q)$$

is cleaned up by the function *fold*, informally:

Test case	t_a	t_i	t_{fast}	t_{comp}	t_{nc}	m_{max}
dsd2html	96	522	884	12	3	24.5M
ontopia2xtm	0.4	2	28	3	2	2.9M
staticanalysis	0.8	2	11	11	8	3.1M
window2xhtml	0.4	1	7	3	2	5.6M
xhtml2fo	48	403	2546	501	493	16.3M
xmlspec	9	12	1437	14	12	22.6M

Table 12: Performance of a select-match compatibility test using abstract evaluation combined with *incompatiblePaths*, vastly superior in time and memory performance to the regular ancestor language test.
t_a is the time used evaluating ϕ_a, t_i is the time used evaluating *incompatiblePaths* and t_{fast} is the time used evaluating ϕ_f (which runs both of the previous).
t_{comp} is the time used performing step 1 of the competition analysis using *completeCoverage*, t_{nc} is the time used performing step 2 of the competition analysis using *completeCoverage* and m_{max} is the maximum heap space used, measured in the same way as for the ancestor language test.

- `child` axis steps immediately followed by `parent` axis steps are both removed from p
- `self::node()` steps are removed from p, as long as that will not make p empty

helping *incompatiblePaths* exclude more infeasible flows, by conservatively rewriting some p to have a "nicer" suffix of `child` steps.

The fast flow algorithm is then:

$$\phi_f(m, i, q, r') = \\ \{q' \in \phi_a(m, i, q, r') | \mathit{incompatiblePaths}(\mathit{fold}(p), \mathit{match}(r'), \{q'\}) = \mathbf{unsure}\}$$

13.13 Template competition analysis

One subject has been swept under the rug until now: Context flows that never materialize at runtime, because they are overridden by context flows to other templates with a compatible mode, a compatible match pattern and a higher import precedence or priority. Needless to say, a flow analysis that does not account for this will be insufficiently precise, and confusing for the user. The subject was deferred to here, because an introduction of it was not necessary for an explanation of the basic flow algorithms, and would indeed just foul it with an even larger wilderness of concepts.

Conservativity must be maintained, meaning that it must be considered very carefully that *all* cases are covered, before a *challenged* edge flow is found to be overridden completely by a higher-priority *challenger* flow: If the challenger flow's target match pattern matches only a subset of what the challenged flow's target matches, for example, there might be some cases where it does not override after all.

In the classic ancestor language flow algorithm, competition analysis was done as:

$match(c)$ is predicate-free, and
$\Sigma_d^*.R(\alpha) \cap \Sigma_d^*.R(match(v)) \cap anl_d(q') \cap (\Sigma_d^*.R(match(c)))^c = \emptyset$

where v is the challenged flow's target, and c is the challenger's target. Both always have the same source and context type. The statement may be read as: For every context node that may be selected by the source, if the flow matches $match(v)$, it also matches $match(c)$. If the challenger has higher priority, all such flows to the challenged target can be deleted; if the challenged flow has a target different from that of the challenger flow, but both have the same priority, a warning about unresolved template competition is issued.

In our effort to remove automata from the flow analysis altogether, there is a shortfall with competition analysis: How can that be done without automata? The answer lies in a dual of the *incompatiblePaths* algorithm, which will be presented next.

A Positive Abstract Evaluation Test

Problem:

Given an XPath path expression S, a pattern M, and a set Ω of DNTs, will M match every node of a type in Ω that S can select? The result is one of **true** or **unsure**, where **true** is conclusive, and **unsure** is not. As *incompatiblePaths*, iteration is over the expressions in reverse order, and so is the step numbering:

Algorithm *completeCoverage*($M \in \mathbf{XPath}_{pattern}, S \in \mathbf{XPath}_{path}, \Omega$):

```
$M \;=\; a_m^m :: t_m^m \cdots a_1^m :: t_1^m$
$S \;=\; a_n^s :: t_n^s \cdots a_1^s :: t_1^s$
  **if** If any step of $M$ has predicates **then**
    return **unsure**
  **end if**
  **if** $M$ has more steps than $S$ $(m > n)$ **then**
    return **unsure**
  **end if**
  **if** $M$ is absolute and $S$ is not **then**
    return **unsure**
  **end if**
  $i \leftarrow 1$; $\Omega_m \leftarrow \Omega$; $\Omega_s \leftarrow \Omega$
  **while** $i \leq m$ **do**
    $\Omega_m \leftarrow S^{test}_{t_i^m}(\Omega_m)$; $\Omega_s \leftarrow S^{test}_{t_i^s}(\Omega_s)$
    **if** $\Omega_s \backslash \Omega_m \neq \emptyset$ **then**
      return **unsure**
    **end if**
    **if** $a_i^m \neq a_i^s$ **then**
    return unsure
  end if
  $\Omega_m \leftarrow R^{axis}_{a^m_i}(\Omega_m)$; $\Omega_s \leftarrow R^{axis}_{a^s_i}(\Omega_s)$
  if $\Omega_s \backslash \Omega_m \neq \emptyset$ then
    return unsure
  end if
  $i \leftarrow i + 1$
end while
return true
```

where

$$\begin{array}{lcl} R^{axis}_{descendant}(\Omega) & = & \mathit{fix}(R^{axis}_{child}(\Omega) \cup R^{axis}_{child}(R^{axis}_{child}(\Omega))) \\ R^{axis}_{descendant-or-self}(\Omega) & = & R^{axis}_{descendant}(\Omega) \cup \Omega \end{array}$$

The idea is to check, step by step from the tail on, whether the M matches *every* node type that S may select. Node tests are applied in the same way as in the normal abstract evaluation, but axis steps are reversed. The test was originally devised for as a fast, lower approximating "stuffer" for the context-sensitive regular ancestor language test (as M in Figure 13.2), thus approximating it both from above and below.

This test performs much better than the ancestor language test, *and* it retains precision, although this time we will not bring a proof, but only state that it is the NFA transition function for *difference* languages. It should be easy to see the truth of this for patterns composed of only `child` and `attribute` axes, and that the general inability of automaton intersection tests to produce anything conclusive for `descendant-of-self` already has been shown.

Given a template-invoking instruction i, a context type q' and two template rules $r_1 \neq r_2$ where $r_2 \geq r_1$, there are two opportunities to conclude that an edge flow from i with context type q must go to r_2 instead of r_1, or, if also $r_1 \geq r_2$, that there is an unresolved competition problem:

1. $\mathit{completeCoverage}(\mathit{match}(r_2), \mathit{match}(r_1), \{q'\}) = \mathbf{true}$, meaning that r_2 will accept any flow that r_1 will.
2. $\mathit{completeCoverage}(\mathit{match}(r_2), \mathit{select}(i), \{q'\}) = \mathbf{true}$, meaning that there may exist ancestors of q that r_1 matches and r_2 does not match, but a node with such ancestors will not be selected. The likelihood of this test returning **true** will increase if path extension is used on the selection.

Removing edges will not jeopardize the general soundness of our fixed point analysis — after removal of an edge flow with context flow q to some template, q is still left in the context set of the template, possibly over-fulfilling the integrity constraint slightly. A simple hack — testing against templates in priority order — can in fact eliminate all removals of established flows, and ensure that competition only eliminates new edge flows, before a context type was otherwise added.

13.14 Precision problems, solutions

Loss of simple text types

The flow analysis presented till now was, with few exceptions, fine for DTD, but it does not sufficiently preserve the simple data types of XML Schema: Having a fine-grained type system for "simple types" (data types), modeling all text nodes as the single node type **pcdata** is too rough an approximation; there is no problem with `value-of` simple-typed declared element node types, but whenever text nodes selected by `child::node()` or by `child::text()` are made edge flow context types, valuable information is lost: Type information is associated not with the text node, but with the element node containing it. Several examples of spurious errors caused by this were found, and, although this was not a very prevalent kind of error, it was considered serious from the perspective of user experience: There should be as few as possible confusing error messages arising from this kind of simple information loss.

The solution employed was simply to *color* the **pcdata** type, subtyping it and parameterize each subtype by an automaton over the type represented, and to regard as different types any two **pcdata** DNTs with different automata. Text nodes from XML Schema complex element types and all DTD `#PCDATA` elements are parameterized by the $\mathbf{char}^*$ language. XML Schema simple-type element declarations q now implicitly spawn a declaration $q' = \mathbf{pcdata}(t)$, with $t = D_d(q) = D_d(q')$

Attributes may be declared with different types in both DTD and XML Schema, but since attribute text is part of the attribute node, and not contained in **pcdata** nodes, attribute declarations need no coloring.

Of course, this subtyping solution will heighten the lattice by (at most) one for each simple type declared, but no significant performance penalty was observed for any practical example.

Processing instructions

Several stylesheets triggered a number of spurious error messages about priority conflicts among template rules matching processing instructions. Really, there were none, as their `match` patterns were all on the `processing-instruction(`*target*`)` form, being mutually exclusive. The problem was solved by declaring, on the fly, one subtype of **pi** for each distinct target name, while still considering all subtypes of **pi** to be compatible with the targetless `processing-instruction()` item test.

Test case	t_{f2}	m_2	t_{f1}	m_1	m_g	$m_{D_{in}}$
dsd2html	884	24.5M	6946	136M	13M	2.7M
ontopia2xtm	28	2.9M	336			
staticanalysis	11	3.1M	691		0.5M	200k
window2xhtml	7	5.6M	407			
xhtml2fo	2546	16.3M	258838	80M		
xmlspec	1437	22.6M	150057	64M	20M	110M
identity/dsd2	1315	23.1M	581491			

Table 13: Time and memory performance of our "fast" flow algorithm compared to that of [23]. t_{f2} and m_2 are flow analysis time and maximum Java allocated heap space for our algorithm, and t_{f1} and m_1 are those of [23]. m_g and $m_{D_{in}}$ are approximate memory sizes of our computed flow graphs and our *automaton* input schema structures for the context-sensitive regular ancestor language flow algorithm, respectively (without automata, input schema structures consume very little memory).

13.15 Conclusion

The final flow algorithm used became a composite fixed point algorithm:

1. A context-insensitive, mode-sensitive fixed point algorithm with $\phi_m \cap \phi_i$, on a fully connected control flow graph
2. A context-insensitive, mode-sensitive fixed point algorithm with ϕ_l, doing a primitive template competition analysis, on the result of the above
3. A context- and mode sensitive fixed point algorithm with ϕ_f, performing the template competition analysis of Section 13.13, on the result of the above

We have reduced time and memory complexity of XSLT flow analysis considerably — from hardly acceptable 110 seconds for the xmlspec test case ([23] clocked 150 seconds) to 1.4 seconds, and from about 260M to 22.6M of memory. There is no loss of precision, or any other weakness of the new approach that we know of. In fact, precision is substantially better than [23], with multiple **pcdata** types, *and* the full class of single-type languages is supported.

Tests on the whole collection of test cases used by [23] showed that both algorithms produced *identical* flow graphs.

Comparing the results of running the largest test cases:

- *XSLT real-time flow analysis is now practical even for large instances.*
- *XSLT flow analysis now works for XSLT2.*
- *XSLT flow analysis now works with XML Schema, without approximation to local-type.*

Part IV

Flow graph applications

A number of different algorithms using the output of the flow algorithm and each contributing to a part of the XSLT editor solution are presented, with special emphasis on solving the static XSLT validation problem for XSLT2 and XML Schema; secondly, the somewhat simpler *code-assist* algorithms.

14 XSLT Validation

Static XSLT 2.0 validation is the prime application of our flow analysis result.

Our XSLT validation works with a formalism modeling XML document generation: *summary graphs*. In XSLT2 stylesheet validation, two such summary graphs are constructed, one generating the possible outcome of the transform, given that input documents are valid wrt. the input schema, and one generating the language of the output schema. The former is then validated against the latter, using an existing algorithm.

14.1 Summary graphs

[48] The definition of summary graphs presented here is a version made for the XACT Project and for the present project[49].

Summary graphs can be thought of as "abstract DOM trees": They abstractly *generate* DOM-like trees rather than *being* trees. We will not describe summary graphs in details here, since they were invented and detailed on elsewhere[50], but we bring a brief reiteration:

A summary graph is defined relative to an input schema D_{in} and an XSLT stylesheet T (the output schema is not used yet). Let $N_{\mathcal{E}}$, $N_{\mathcal{A}}$, $N_{\mathcal{T}}$, $N_{\mathcal{S}}$, and $N_{\mathcal{C}}$ be sets of **element** nodes, **attribute** nodes, **text** nodes, **sequence** nodes, and **choice** nodes, respectively.

Intuitively, the former three kinds of nodes represent the possible elements, attributes, and character data or attribute values that may occur when running the stylesheet on input valid wrt. to D_{in}. The **sequence** and **choice** nodes are used for modeling element contents and attribute lists. The edges in the graph describe how the constituents can be composed to

[48]This summary graph introduction is largely a copy the work from [20], provided by Anders Møller. It is brought mostly to provide a complete introduction to summary graphs.
[49]by Anders Møller, BRICS
[50]

form XML documents. More precisely, a *summary graph* SG is a tuple

$$SG = (N_{\mathcal{E}}, N_{\mathcal{A}}, N_{\mathcal{T}}, N_{\mathcal{S}}, \mathcal{N}_{\mathcal{C}}, R, S, \mathit{contains}, \mathit{seq}, \mathit{choice})$$

where

- $N_{\mathcal{E}}$, $N_{\mathcal{A}}$, $N_{\mathcal{T}}$, $N_{\mathcal{S}}$, and $N_{\mathcal{C}}$ are finite disjoint sets of *nodes* of the different kinds mentioned above; for later use we define $N = N_{\mathcal{E}} \cup N_{\mathcal{A}} \cup N_{\mathcal{T}} \cup N_{\mathcal{S}} \cup N_{\mathcal{C}}$;
- $R \subseteq N$ is a set of designated *root nodes*;
- $S : N_{\mathcal{E}} \cup N_{\mathcal{A}} \cup N_{\mathcal{T}} \rightarrow 2^{\mathbf{char}^*}$, defines *node labels*;
- *contains* : $N_{\mathcal{E}} \cup N_{\mathcal{A}} \rightarrow N$ defines *contains edges*;
- *seq* : $N_{\mathcal{S}} \rightarrow N^*$ defines *sequence edges*; and
- *choice* : $N_{\mathcal{C}} \rightarrow 2^N$ defines *choice edges.*

The *language* $\mathcal{L}(SG)$ of a summary graph SG is the set of XML trees that can be obtained by unfolding the graph, starting from a root node:

$$\mathcal{L}(SG) = \{x \mid \exists r \in R : r \Rightarrow x\}$$

We here use the *unfolding relation*, $\Rightarrow$, between summary graph nodes and XML trees, which is defined inductively as follows:

$$\frac{n \in N_{\mathcal{E}} \quad e \in S(n) \quad \mathit{contains}(n) = m \quad m \overset{\mathsf{attr}}{\Rightarrow} \{a_1, \ldots, a_k\} \quad m \overset{\mathsf{cont}}{\Rightarrow} c}{n \Rightarrow \texttt{<}e\ a_1 \ldots a_k\texttt{>}\ c\ \texttt{</}e\texttt{>}}$$

$$\frac{n \in N_{\mathcal{A}} \quad a \in S(n) \quad \mathit{contains}(n) = m \quad m \overset{\mathsf{text}}{\Rightarrow} s}{n \Rightarrow a\texttt{="}s\texttt{"}}$$

$$\frac{n \in N_{\mathcal{T}} \quad s \in S(n)}{n \Rightarrow s}$$

$$\frac{n \in N_{\mathcal{S}} \quad \mathit{seq}(n) = a_1 \ldots a_k \quad a_i \Rightarrow b_i \text{ for all } i = 1, \ldots, k}{n \Rightarrow b_1 \ldots b_k}$$

$$\frac{n \in N_{\mathcal{C}} \quad a \in \mathit{choice}(n) \quad a \Rightarrow b}{n \Rightarrow b}$$

This definition uses the operations $\overset{\mathsf{attr}}{\Rightarrow}$, $\overset{\mathsf{cont}}{\Rightarrow}$, and $\overset{\mathsf{text}}{\Rightarrow}$ to extract attributes, contents, and text. These relations are defined as follows:

$$\frac{n \in N_{\mathcal{A}} \quad n \Rightarrow a}{n \overset{\text{attr}}{\Rightarrow} \{a\}} \qquad \frac{n \in N_{\mathcal{E}} \cup N_{\mathcal{T}}}{n \overset{\text{attr}}{\Rightarrow} \emptyset}$$

$$\frac{n \in N_{\mathcal{S}} \quad seq(n) = a_1 \dots a_k \quad a_i \overset{\text{attr}}{\Rightarrow} A_i \text{ for all } i = 1, \dots, k}{n \overset{\text{attr}}{\Rightarrow} \bigcup_{i=1}^{k} A_i}$$

$$\frac{n \in N_{\mathcal{C}} \quad a \in choice(n) \quad a \overset{\text{attr}}{\Rightarrow} A}{n \overset{\text{attr}}{\Rightarrow} A}$$

$$\frac{n \in N_{\mathcal{E}} \cup N_{\mathcal{T}} \quad n \Rightarrow x}{n \overset{\text{cont}}{\Rightarrow} x} \qquad \frac{n \in N_{\mathcal{A}}}{n \overset{\text{cont}}{\Rightarrow} \epsilon}$$

$$\frac{n \in N_{\mathcal{S}} \quad seq(n) = a_1 \dots a_k \quad a_i \overset{\text{cont}}{\Rightarrow} b_i \text{ for all } i = 1, \dots, k}{n \overset{\text{cont}}{\Rightarrow} b_1 \dots b_k}$$

$$\frac{n \in N_{\mathcal{C}} \quad a \in choice(n) \quad a \overset{\text{cont}}{\Rightarrow} b}{n \overset{\text{cont}}{\Rightarrow} b}$$

$$\frac{n \in N_{\mathcal{T}} \quad n \Rightarrow x}{n \overset{\text{text}}{\Rightarrow} x} \qquad \frac{n \in N_{\mathcal{E}} \cup N_{\mathcal{A}}}{n \overset{\text{text}}{\Rightarrow} \epsilon}$$

$$\frac{n \in N_{\mathcal{S}} \quad seq(n) = a_1 \dots a_k \quad a_i \overset{\text{text}}{\Rightarrow} b_i \text{ for all } i = 1, \dots, k}{n \overset{\text{text}}{\Rightarrow} b_1 \dots b_k}$$

$$\frac{n \in N_{\mathcal{C}} \quad a \in choice(n) \quad a \overset{\text{text}}{\Rightarrow} b}{n \overset{\text{text}}{\Rightarrow} b}$$

The language of summary graphs are of an even greater class than general tree languages.

Notice a few detail differences between DOM and summary graphs: XML attribute nodes to not contain text nodes. Summary graph attribute nodes have edges to text nodes, though. Summary graphs do not model the **comment** and **pi** node kinds, as these are hardly worth consideration in a validity assessment context: They are permitted almost anywhere anyway[51].

There are a few limitations applying to legal summary graphs: **interleave** nodes may only appear as the content node of an element node, and element content models must be single-type.

[51]In DTD, they are not permitted as children of element declared **EMPTY**, but we have never heard of any major disasters caused but somebody overlooking that. We speculate that it was simply forgotten to explicitly permit them in the XML spec.

Summary graph validation

An existing algorithm in BRICS Schematools can check, for any two summary graphs (SG_1 and SG_2), whether $\mathcal{L}(SG_1) \subseteq \mathcal{L}(SG_2)$

In the negative case, a list of error messages is produced, listing every element content model whose constraints are violated, with an example of a violation where possible.

We will simply use this algorithm for validation: The summary graph constructed to model out transform output, given that transform *input* is valid wrt. D_{in} is SG_1, and the *output* schema will be used as a basis for constructing SG_2.

14.2 Summary Graph construction

Having constructed a flow graph, we will use the information of that for constructing a summary graph, generating a superset of the XML tree set that the given transform may generate for D_{in}. At this stage, the data description goes from *monovariant* (one graph node for each template rule and template-invoking instruction, template nodes are decorated with context sets and edges with context flows) to an equivalent *polyvariant* graph — there is now one node for each $(\mathbf{templaterule}, \mathbf{contexttype})$ pair, and no context set decorations on nodes or edges.

The summary graph will extend the flow graph in two dimensions, from describing only flow of control and context types between template rules during stylesheet invocation, it will describe everything missing in the flow graph alone:

- The nodes *output* during stylesheet invocation will be modeled. The very reason for making the summary graph polyvariant is to support this, by making available a specific context type for each sequence constructor.
- The *content models* of nodes, as given by the input schema, were approximated coarsely in the flow analysis: All information in the content and attribute models of DNTs was thrown away in the flow analysis, except whether some type was maybe a child or an attribute of another. Flows were modeled on the large side, and never deemed not to exist, unless that could be established with absolute certainty. The summary graph constructed will to a large extent compensate for that, re-establishing the limits and the "maybes" ignored earlier — and still conservatively: The graph reflect any uncertainty that some flow or output will actually happen at transformation run-time.

  The interplay between the flow analysis and the summary graph construction can loosely be put this way:

  – Flow analysis takes care to model any run-time invocation of templates that may cause *too much* output for validity.
  – The summary graph is composed from *fragments*, that are assembled on the basis of information from the control flow graph.

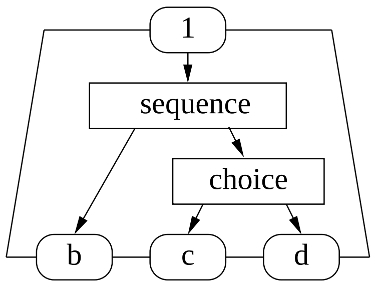


Figure 21: A summary graph fragment with placeholders for the DNTs b, c and d. It is typical for a content model fragment, having no **element** or other output-generating nodes.

- The individual summary graph fragments model invocations of template instructions, taking care to also model that invocations may *not* happen, causing *too little* output for validity[52].

The goal is to construct a summary graph, given a flow graph and an input schema. The summary graph constructed must generate a tight superlanguage of the language generated by the stylesheet, given that input is valid wrt. the input schema D_{in}

The process of converting the graph is somewhat involved, and in the demonstrator implementation, it accounts for the greater part of the source code. Despite appearing complex at places, the time and space complexity of summary graph construction are just bounded by the number of template rules multiplied by the number of declared node types, where each template rule counts as the number of modes it is used for[53].

The basic building block used is the *summary graph fragment*, which we will describe in a little more detail:

14.3 Summary graph fragments

A *summary graph fragment* is a a summary graph extended by an *entry node* and a mapping from DNTs to *placeholders*. Placeholder maps serve to allow specific **choice** nodes of a summary graph fragment to be associated with DNTs. Summary graph fragments, although very simple, are capable of filling out these roles:

- Modeling evaluation of XPath2 step expressions, from a given context and restricted by a schema
- Modeling XSLT2 sequence constructors and template-invoking instructions

[52]This perception is not quite complete: A template may contain some instruction that outputs, say, an undeclared element, which is really a "too much". A template rule may have its in-flow sucked up by another template, or it may be missing altogether, causing a "too little". But for a three-item summary, it will do.

[53]So, unless the #**all** mode is used, a template is just a template.

- Cloning schema content models

In overview, summary graph construction consists of these steps:

1. Fragment construction: For each mode $m \in \mathcal{M}$, for each template rule $r \in \mathcal{R}$, for each context type $q \in C(r)$, a fragment $f_r(m, r, q)$ is constructed, representing r when invoked under mode m, with q as the context type.

   The fragment will model the output of r's sequence constructor as faithfully as practical. Each template-invoking instruction $\{i : r_i(i) = r\}$ has a separate fragment constructed, which may be referenced later as $f_i(m, i, q)$; this fragment will embody information about the nodes that i may select, under the restrictions of the input schema.

2. Fragment assembly: The information stored in the flow graph is used for assembling the fragments into a connected summary graph, representing the possible output of the entire transform. This final summary graph is then validated against another summary graph representing the output schema.

14.4 Modeling sequence constructors

The summary graph fragments elaborated on here model the output and outgoing flow of some template rule *once invoked* — Assume that the template rule r is invoked with the context type q, under the current mode m. The fragment representing this will be roughly isomorphic with the sequence constructor of r, and it is appropriately transformed in a recursive descent over this:

Sequences of instructions are transformed to a **sequence** node, with the children being the transforms of the sequence's children, in the same order

`value-of` **instructions** are transformed to a **text** node, with a text language automaton constructed as detailed in Section 14.4.

`attribute` **instructions** are transformed to an **attribute** node, with a name automaton constructed as in Section 14.4.

`element` **instructions** are transformed to an **element** node, with a name automaton constructed as in Section 14.4.

`choose` **instructions** are transformed to a **choice** node, with a *choice* edge to the transform of each child

`when` **instructions** are transformed to the transform of their child sequence

`otherwise` **instructions:** are transformed to the transform of their child sequence

`copy` **instructions** are transformed depending on q:

Element context types result in an **element** node, constructed in the same way as from an `element` instruction.

Attribute context types result in an **attribute** node, constructed in the same way as from an `attribute` instruction. A **text** node is added as content, having the language $D_d(q)$, as defined in Section 6.8.

pcdata(l) [54] context types result in a **text** node, with the language l.

The root context type results in the transform of the child instruction sequence of the `copy` instruction.

The comment context type results in a **sequence** node with no edges.

The pi context type results in a **sequence** node with no edges.

The translation of any content of the `copy` instruction is made content of its translation.

Template-invoking instructions are transformed to a rather involved structure, containing at least one placeholder for each DNT that may be selected by the instruction, and a model that describes, using input schema information, how many times each DNT may be selected may be selected, and in what order relative to other DNTs. This is described separately in Section 14.5.

`value-of` INSTRUCTIONS

This instructions always evaluates to a text node, and it is translated to a **text** node[55]. A best-effort attempt is made to bound its language to what the `select` expression may evaluate to:

union expressions result in the union language of the languages of each sub-expression

`concat(...)` function calls result in the concatenation language of the argument languages

`name()` function calls result in the singleton language of the name of the context type[56]

`local-name()` function calls result in the singleton language of the local name of the context type

`namespace-uri()` function calls result in the singleton language of the namespace URI of the context type

string literals result in the singleton language of themselves.

[54] The coloring of **pcdata** was explained in Section 13.14.

[55] As one might have guessed.

[56] The function is specified as a string function, not a qualified name function. This leaves a small problem as to which namespace prefix to use: Using the language over all **NCNAME**s would trigger any error with the value as a datatype (such as not matching an enumerated simple type), but relying on that for use in `<element name="name()"/>` would be a precision disaster. The solution is to use the **NCNAME** language in a `value-of` evaluation context, and use a prefix bound to the namespace of the context node otherwise.

`position()` and `last()` function calls result in a language over the integers from 1 up, lexically.

all other function calls are currently given up upon, and result in the **char*** language. There is room for some improvement here, since XPath2 functions are strongly typed, and some can be approximated to something better than just their declared type, like with `concat()`. The only reasons that this was not pursued any further was the time it would take to write a full function and operator library including coercion rules for XPath2, and that nothing indicated a strong need for it.

path expressions are abstractly evaluated to a set of DNTs. These are each approximated to a regular language that contains the results of evaluating `<value-of select="."/>` on each type:

- **pcdata(l)** evaluate to the language l.
- **DTD-declared element types** evaluate to the language of the empty string if declared **EMPTY**, otherwise to the **char*** language.
- **XSD-declared complex-typed mixed element types** evaluate to the **char*** language.
- **XSD-declared complex-typed non-mixed element** evaluate to the empty-string language if childless, otherwise to to the language of all XML strings (giving up precision — `value-of` on an instance element returns the concatenation of all its descendant text nodes, and is hard to limit statically)
- **XSD-declared simple-type or simple-content element types** are not as easy to deal with as it may seem: XSLT2 *atomizes* their value, splitting instances of some non-primitive XDM types into a sequence of the corresponding, primitive atom type, and normalizing whitespace. These atoms are then output separated by whatever is given in the `separator` attribute of `value-of`, defaulting to a `#&x20;` (a space).

  As long as the `separator` attribute is absent, the language of the element's declared type is bound to be a superlanguage of that of the `value-of`, so that can be safely used. Currently, if the attribute is present and has a value that is not allowed as a separator for list-derived simple XML Schema types, **char*** is used as an approximation. Of course, looking up the atom type's language l in the schema, and constructing a list language based on l, `separator` and the length bounds of the list derivation would be slightly better.
- **Attribute types** evaluate to the language of their fixed value if present, otherwise as with simple-type or simple-content element types.
- **the root, comment and pi types** evaluate to **char***.

`element` AND `attribute` INSTRUCTIONS

These instructions have a `name` and a `namespace` attribute value template (AVT), from which, together with the context node, an automaton is constructed that has as its language a superset of the names of the elements that the instruction may construct.

First, the `name` AVT is parsed into a sequence $\{e_1, \cdots, e_n\}$ of XPath expressions in the same way as described in Section 12. A function invocation expression `concat`$(e_1, \cdots, e_n)$ is constructed, and an automaton a_{name} is constructed as for the `value-of` instruction. The same is done for the `namespace` AVT ($a_{namespace}$), and the two automata are then, obeying the conflict resolution and no-prefix namespace semantics specified for the `element` cf. the `attribute` instruction, merged to form an automaton whose language is:

$$\begin{cases} '\{'.\mathcal{L}(a_{namespace}).'\}'.\mathcal{L}(a_{name}) & \text{if } \mathcal{L}(a_{namespace}) \neq \{\epsilon\} \\ \mathcal{L}(a_{name}) & \text{otherwise} \end{cases}$$

which is the “Clark notation” of expanded qualified names expected by BRICS Schematools.

Care has been taken to take advantage that the XPath2 functions `name`, `local-name` and `namespace-uri` can be evaluated statically because a context type is available: The generic identity transform thus did not have any precision problems with any schemas tried.

14.5 Modeling template-invoking instructions

A template-invoking instruction is an instruction that selects a sequence of nodes, and then, for each node in the sequence, passes control to a template matching the node. An `<apply-templates select="`s`"/>` instruction has an XPath2 expression s for selecting to node sequence, which is specified in [15] to be erroneous if it can select anything else than nodes. The other two template-invoking instructions left after stylesheet simplification, `apply-imports` and `next-match`, always select the singleton sequence of the context node, and are treated uniformly with the `apply-templates` instruction by adding `self::node()` as a selection expression.

First, let us get rid of those `apply-templates` instructions that were made to select `xslv:unknownSequence()` in stylesheet simplification. They result in a fragment outputting an element with any name at all, enough to set off a somewhat meaningful error message in validation.

Each template-invoking instruction is translated into a summary graph fragment. The idea is to compose the fragment as an abstract evaluation of the selection expression of the instruction, from a chain of *step fragments*, of which one is constructed for each each location step of the selection expression.

If s is on the form p_1 `intersect` p_2 or p_1 `except` p_2, we model selection of some unknown subsequence pf what p_1 may select, translating only for p_1 and letting the result be a **choice** node over the translation and an empty **sequence**.

Step fragments are constructed differently, depending on the axis of the source step. In any case, the construction starts given set Ω_i of context types — the first step in a selection expression is constructed given the context node q for the summary graph fragment construction; $\Omega_0 = \{q\}$.

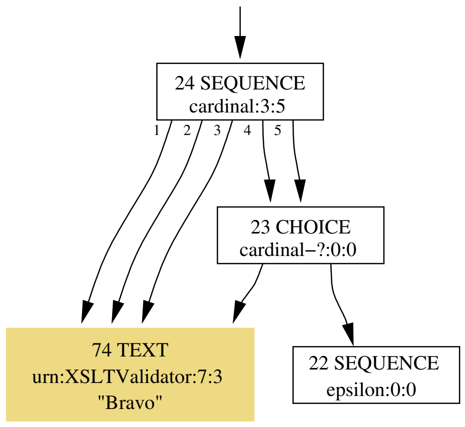


Figure 22: Cardinality construction: 3-5 times the text "Bravo".

The result of evaluating abstractly the single step is constructed:

$$\Omega_{i+1}S(a :: t, \Omega) = S^{step}_{a::t}(\Omega_i)$$

For multi-step selections, a summary graph fragment is constructed for each DNT in Ω_{i+1}, and the resulting fragments are concatenated onto the fragment constructed from the previous step.

The fragment construction depends on the axis of each step, and the context type:

child: The content model of the DNT is modeled in a recursive descent using summary graph primitives:

simple-typed elements are modeled as **pcdata**(t) placeholder, where t is the language of $D_d(t)$ if there is a text type $\in \Omega_{i+1}$, otherwise as an empty **sequence** node.

mixed-complex-typed elements: If there is a text type $\in \Omega_{i+1}$, the result is a **interleave** node, with a placeholder for **pcdata**(**char***) as one child, and the model of the type's other contents as another child. Otherwise, the result for the same complex type, but non-mixed, is returned.

nonmixed-complex-typed elements are modeled as the model of their outermost compositor.

Sequence compositors are modeled as a **sequence** node, recursing over children.

Choice compositors are modeled as a **choice** node, recursing over children.

All compositors are modeled as an **interleave** node, recursing over children.

element declarations $q \in \Omega_{i+1}$ are modeled as a placeholder for q.

Card constructs are modeled precisely, using the **sequence**, **choice** and **oneOrMore** summary graph node types.

element declarations not $\in \Omega_{i+1}$ are modeled as an empty **sequence**.

any wildcards are modeled as a cover-any-case construct representing the *result* of transforming the wildcard — outputting any elements, attributes or text in any order, with the construct itself as the content model of elements and any text as the model of attributes and text nodes.

comment and pi if $\in \Omega_{i+1}$, are modeled as a placeholder for the **comment** DNT and one for the **pi** DNT.

finally, a cleanup is performed:[57],

Dead subgraphs of **sequence**, **choice**, **interleave** and **oneOrMore** that do not have a reachable placeholder are removed. Such subgraphs may arise if the set Ω_{i+1} does not contain every DNT of the content model. This is repeated until no dead subgraphs are left.

Singleton sequence, **choice** or **interleave** nodes are replaced by their content.

parent: A fragment is constructed over a **choice** node, branching to each of $\{q' \in \mathbf{parent}(q) | q' \in \Omega_{i+1}\}$.

attribute: A **sequence** node[58] is returned, with the model of each attribute use, and attribute wildcard of the context type as children:

attribute uses in Ω_{i+1} are modeled as a placeholder node mapping the attribute's DNT. If the cardinality of the attribute use is **Optional**, an optional construct using **choice** is added.

`anyAttribute` **wildcards** are modeled the same way as `any` (the rationale being that attributes, too, may cause invocation of templates that output elements — it is the transform *output* being modeled).

all other axes A fragment is constructed over a **choice** node, branching to a placeholder for each DNT in Ω_{i+1}. Admittedly, a bit more precision could be achieved by using the axis directions: For example, on the **ancestor** axis, it is known that a near ancestor comes before a remote ancestor — unless the ancestor types are mutually recursive.

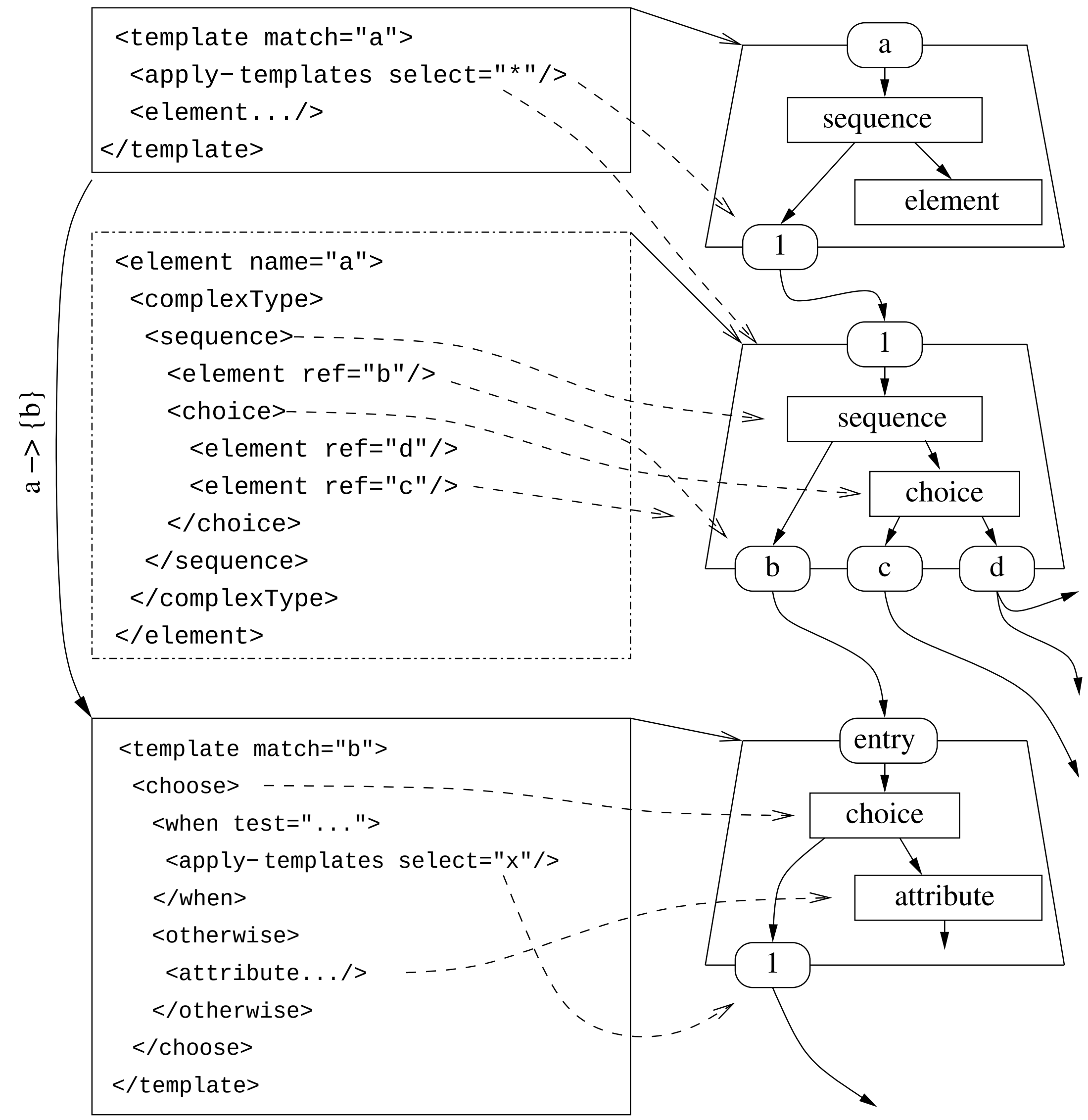
A general approach could be using the Mohri-Nederhof algorithm to approximate document order, given a schema. We have seen one case in the [23] test corpus (selections beginning with `/descendant-or-self::node()/...`) that could benefit from that — on the other hand, the expressions could as well (automatically?) have been rewritten to something sharper.

A union selection problem and solution

In the above description, we have described context flows as emanating directly from template-invoking instructions, and glossed over union selection expressions completely. Union expressions in selections are handled by considering each template-invoking with a union selection

[57]"finally" is conceptual — in an implementation, the cleanup may be done on the fly, during construction

[58]Attributes are unordered, so the order of the attributes on the **sequence** node does not matter. **interleave** could have been used, but it is costlier performance-wise.



```
<template match="a">
 <apply-templates select="*"/>
 <element.../>
</template>
```

```
<element name="a">
 <complexType>
  <sequence>
   <element ref="b"/>
   <choice>
    <element ref="d"/>
    <element ref="c"/>
   </choice>
  </sequence>
 </complexType>
</element>
```

```
<template match="b">
 <choose>
  <when test="...">
   <apply-templates select="x"/>
  </when>
  <otherwise>
   <attribute.../>
  </otherwise>
 </choose>
</template>
```

Figure 23: Constructed and assembled summary graph fragments. The upper fragment is translated from a template rule with an `apply-templates` instruction. The center fragment is the translation of the content model of the `a` element, and the fragment at the bottom is one of the targets of the original `apply-templates` instruction.
The upper and center fragments are assembled already at fragment construction time; the lower fragment was connected using the flow graph information when the upper template is invoked with an-a typed node, there is an edge flow of node type b to the lower.

expression a set of template-invoking instructions, each with a singleton path selection expression. When constructing summary graph fragments for union selection template-invoking instructions, care must be taken not to introduce any new ordering: Nodes are selected in document order, even by a union selection expression.

A conservative approach would be to represent such a group by a **OneOrMode** node containing a **Choice** node over the individual instructions' summary graph fragments; call this the "emergency union resolution". This certainly does not mask any errors there may be in ordering of output from the fragments; but the **Choice** node causes a rather bad loss of precision, saying basically that all fragments but one is now only *maybe* invoked.

Most union selection expressions are on the form `child::a | child::b | ...`, selecting more than one, but fewer than all child elements of the context node. This is simply restructured to `child::DNTs(`$a, b, \cdots$`)`, where a, b etc. are sequences of all element DNTs that have the name `a`, `b`, etc. Now, the union selection expression has become a single path expression, semantics preserved. This "hack" eliminated a spurious error in the links2html test case. The same technique is used on unions of `attribute`-only axis one-step path expressions.

For union expressions with different first-step axes, an approximation better than emergency union resolution was used:

1. Call the set of path expressions of the union $P = \{p_1 \cdots p_n\}$.
2. Let P_{before} be the set of all relative $p \in P$ with a leading `parent` or `ancestor` axis step, followed by zero or more `parent` or `ancestor` axis steps.
3. Let P_{self} be the set of all $p \in P$ with a single `self` axis step.
4. Let P_{attr} be the set of all $p \in P$ with a single `attribute` step.
5. Let P_{after} be the set of all relative $p \in P$ with a `child` or `descendant` steps, not followed by any `parent` or `ancestor` steps.
6. If $|P_{before}| + |P_{self}| + |P_{attr}| + |P_{after}| \neq n$, then give up and use emergency union resolution on P.
7. Otherwise, make a list of $\{P_{before}, P_{self}, P_{attr}, P_{after}\}$ with emergency union resolution applied to each, except empty ones, which are removed, or singletons, which are added to the list directly. The result is a **Sequence** node over the list.

This heuristic conservatively re-orders the path expressions of the union to the document order of what they select, but it gives up in the face of absolute expressions, `ancestor-or-self` or `descendant-or-self` first steps or path expressions of mixed forward and reverse order steps.

Finishing touches

In case one or more `sort` instructions are nested in an `apply-templates` instruction modeled, at runtime, the node sequence selected will be re-ordered between selection evaluation and template recursion, and any information about ordering of DNT flows that the constructed summary graph fragments contain may become invalid. This may affect soundness of the analysis (as masking an illegal ordering of output, caused by `sort` reordering) if not dealt with.

The solution used was to replace **sequence** nodes by **interleave** nodes, preserving cardinality but breaking ordering, but with one restriction: BRICS Schematools can only handle **interleave** as the direct content of **element** nodes, so if if it is not known for sure that the **sequence** node will have that position, an **oneOrMore** over a **choice** node is used instead, sacrificing precise cardinality but retaining soundness.

One often used predicate is the integer literal `[`i`]`, introduced in Section 8.1. It will filter out all but the i'th node in a sequence. If used in the last step of a selection expression, its effect is that at most one node is selected (the i'th, or, if there are not i nodes in the sequence, none). Selection expressions with this condition can have their summary graph fragments constructed more precisely, by effectively limiting Card expressions to $max = 1$. It is done at fragment construction time.

All *predicates* in selection expressions can potentially preclude a node from being selected, and thus cause the transform to get invalid, because any output caused by that selection was missing. The effect of the predicate is modeled by replacing the translation of the `apply-templates` instruction by a **choice** node, choosing either the translation or an empty **sequence**.

A further refinement that was left as an idea for future work is a way of bounding the types selectable by a `[`i`]` filter expression: Construct a deterministic automaton over $C_m(q)$ (it is regular), and remove all dead states. For the number i, the set of symbols that may appear at the i'th position of a string generated by the automaton (and thus be the DNT of the i'th node in a string; the node selected), perform a depth-i traversal into the DFA graph of the automaton, starting from an entry node. A set of states will be reached. The set of symbols of all out-edges from states in the state set is the set of possible i'th symbols in the language, and it might narrow down Ω_{i+1}.

14.6 Fragment assembly

Summary graph assembly is the stage of validation where the flow graph and the constructed summary graph fragments finally unite: We have constructed one (composite) fragment for each template, with a set of placeholders for each union subexpression of each contained template-invoking instruction. All that remains to be done is an assembly of everything, using the edges of the flow graph for inserting edges.

Recall the flow graph $G = (F, C)$. Assembly is as simple as:

For all $m \in \mathcal{M}, q \in \Sigma_d, i \in \mathcal{I}, r \in \mathcal{R}$:

For all $q' \in F(m)(q)(i)(r)$:

Add a *choice* edge from the placeholder for q' at $f_i(m, i, q)$ to $f_r(\mathit{modeflow}(m, \mathit{mode}(i)), r, q')$.

This is easily implemented as a work-list algorithm.

15 Evaluation of validation

D'oh!

Homer Simpson

Finding a proper way to evaluate the validator was harder than expected: It is *very* hard to find working empirical cases of XSLT2 and/or XML Schema based transform *triples* of an input schema, a transform and an output schema.

Since performance-related aspects are almost the same for DTD- and XML Schema based input, we have simply re-utilized previous work — the [23] set of test triples, believed to have been originally provided by Michael I. Schwartzbach.

15.1 DTD-based test set

Since the [23] XSLT static validation project already has demonstrated that static XSLT validation for DTD and XSLT1 is feasible with a quite acceptable precision, and since our algorithm improves upon the previous one in all respects, we will not need to repeat the basic findings here. Just to be sure, his set of DTD/XSLT1 test cases were run again, on our validator.

The test triples

Both [23] and we implemented in Java ([23] 1.4; we 1.5), and ran all tests on a 3 GHz Pentium 4 computer with 1GB of RAM, and running Linux. All times are in milliseconds.

Validation performance

Performance results for a selection of the [23] DTD-based test set are in Table 15. For once, there is something we have little to say about; the performance of the stylesheet simplifier,

Title	# templates	# element decl.	# attribute decl.
`adressebog2xhtml`	16	11	0
`agenda2xhtml`	11	6	5
`availablesupplies`	10	18	5
`dsd2html`	990	40	15
`emaillist`	54	20	1
`links2html`	20	12	1
`ontopia2xtml`	70	11	13
`order2fo`	14	19	4
`poem2xhtml`	14	7	1
`purchaseorder`	18	22	6
`slides2xhtml`	21	13	2
`sqlprocedures`	20	14	3
`staticanalysis`	59	22	7
`window2xhtml`	102	37	7
`xhtml2fo`	462	89	119
`xmlspec`	384	162	56

Table 14: Complexity of the [23] triples

the input schema structure initialization (which now has no automata to construct), and the flow analysis are *fast*, fast enough to be able to run a cycle every few seconds with any input schema and stylesheet of practical size.[59]. In the two largest cases, flow analysis completes in about 1/100th of [23]'s time, despite better precision and a much lower memory profile.

There should then be no more reason for concern as to whether practical, reasonably precise XSLT2 and XML Schema flow validation is feasible; it *is*.

[59]Probably, this performance could be improved even more by simply turning off compiler debugging instrumentation, and turning on optimization. Also, a simple experiment with bit sets showed that there is a potential for making the many set addition and removal operations in the abstract evaluation algorithm run almost twice as fast, compared to the current hash sets.

Test case	t_{load}	$t_{D_{in}}$	t_{flow}	t_{sgc}	$t_{D_{out}}$	t_{val}	t_{tot}
adressebog	95	5	13	8	880	3646	4647
availablesupplies	95	4	0	8	857	4608	5590
dsd2html	842	13	1642	2175	950	214057	219680
sqlprocedures	140	5	26	31	882	8313	9397
window2xhtml	694	8	118	100	896	15649	17465
xhtml2fo	794	125	1171	-	-	-	-

Table 15: Execution timing, for a selected set of test cases (most typical small cases were omitted). BRICS Schematools seems to hang on the xhtml2fo test case. Overall, simplification and flow performance are all fine for on-line use of the algorithms, but summary graph validation is slow, and had better be run in a separate thread.

Test case	true	spurious	comment
adressebog	1	0	illegal element in output
agenda2xhtml	1	0	required attribute missing
availablesupplies	2	0	missing XHTML `head`, `alt` attribute on `img`
dsd2html	0	0	
emaillist	2	0	two documents elements output; `p` in `small`
links2html	2	0	`table` directly in `p`, `h2` directly in `p`
ontopia2xtm	7	1	`NMTOKEN` attributes emptiable, one `if` test spurious
order2fo	0	0	
poem2xhtml	1	0	missing XHTML `head`
purchaseorder	1	1	one // step 0-infinite cardinality estimate spurious error
slides2xhtml	11	2	complicated depth-4 recursive error, possible flow analysis bug. 9 errors are missing attributes
sqlprocedures	6	0	`color` attributes on `hr`, emptiable `ul`, `ol` content
staticanalysis	1	0	required attribute missing
window2xhtml	1	0	text before document element
xhtml2fo	-	-	summary graph validation not terminated
xmlspec(DTD)	-	-	summary graph validation not terminated

Table 16: True and spurious errors for most DTD and one XSD test cases.

Summary graph validation, however, has become *slower* in the larger cases, and in the DTD-xhtml2fo case, does not terminate at all within reasonable time (hours). Now this could be due to BRICS Schematools still rather immature, or it could be that we generate bad summary graphs. Running a Schematools check on the generated summary graphs revealed nothing, though, and an inspection for zero-out-degree **choice** nodes — a case we were advised against — found none.

We hope that this is only an intermittent situation with BRICS Schematools (or that it is a bug in our summary graph construction, for that matter), and believe that it can be solved. At least, [23] has showed that fast summary graph validation is possible — his dsd2html summary graph validated in less than a minute. Maybe some automaton uses in BRICS Schematools can be replaced by something faster, following the idea of our flow analysis.

Precision

Precision-wise, the envelope of static analysis was once again being pushed.

The data here does not compare directly with that of [23], because of differences in reporting counts of the same errors in the different summary graph validation software used. However, both in theory and practice, we are more precise: A problem with attribute datatypes being approximated to **char*** caused [23] a number of spurious errors — we have almost-exact data types.

As for these DTD-based typical cases of schema/stylesheet triples, there remains only to say that precision is excellent, and that our proposed Mohri-Nederhof document order approximation, and some kind of predicate evaluation (like our drop-off decorators) seem to be the only things missing for perfection — with this test set, that is.

XSD based test cases

Empirical testing with XML Schema bases triples was a moderate success, primarily because available output-side schemas would not load into BRICS Schematools at the time of writing, and could not be converted reliably to something that would.

Quite late in the process, we found a solution, but by then there was no time for evaluation of a larger, real-world test set. This remains, then, largely a to-do.

Generally

Demonstrating anything empirically about the adequacy of our modeling of XSLT2-specific constructs was not possible, as it is not even a W3C Recommendation yet, and there is currently only few and expensive schema-aware stylesheet processors available. We have instead resorted to self-made unit test cases, on the aspects that we believed were the most important:

- Proper functioning of the stylesheet simplifier; no masking of errors
- Proper behavior for modes and `apply-templates`/`apply-imports`
- Proper behavior for modes in summary graph construction
- Identity transformation error-free with copying of any simple datatype
- Identity transformation error-free with same-name, different-type elements in schema
- Identity transformation error-free with substitution groups in schema
- Identity transformation error-free with subtyping by extension in schema
- Reuse of experience: Identify corrections for [23] spurious errors

The tests are not automatic regression tests[60]; they were run manually, and the results (an XML dump of a control flow graph, or a summary graph) were inspected manually.

[60]For a production XSLT tool, a set of regression tests would be a *must*: Errors can come a long way through data structures, and backmapping their origin can be very time-consuming. For this project, which merely investigates feasibility, a test suite is too expensive.

Validation

Instead, we have focused on finding any problems in the transition from XSLT Version 1 to Version 2, and from DTD to XML Schema. Ignoring issues related to XSLT2 parameters and variables (a full implementation of flow analysis on temporary trees or `copy-of` temporary trees does not yet exist), the primary source of new spurious errors is the increased expressiveness/strictness of XML Schema types.

How many spurious errors can be found depends strongly on the *quality of the output schema used*. Throughout the project, proper evaluation was seriously hampered by:

- Scarcity of empirical test cases (triples of transforms and input, output schemas)
- Scarcity of high quality output schemas

Empirical test cases were *hard* to find on the Web: Many were not proper XML to XML transforms, and many lacked a working input schema. This situation does not, however, mean that the whole idea of static XSLT validation in is still-born for real-world application: As soon as the XSLT Version 2 transition is complete, stylesheet authors will become aware of the coupling between schemas and stylesheets (all will be needed at the same time), and they will become more likely to be distributed together.

The quality problem of output schemas had a simple cause: Through the part of the project where BRICS Schematools was available[61], it would only construct summary graphs from one schema language: A restricted subset of Relax NG. Not a single Relax NG schema found on the Web was on the proper, restricted form, and the only converter found to convert XML Schema to Relax NG (Sun's rngconv) would either crash (for all XML Schema versions of XHTML1 schemas found), or convert to something that was not on the proper, restricted form. Conversion from DTD (still with rngconv) was, then, used as a last resort, resulting in output schemas that did not exceed DTD in expressiveness.

An almost obvious solution was found in the final hour[62]: To use as the output schema summary graph the *transform* summary graph constructed from an *identity* transform, using the original *output* schema on the *input* side. This is quite feasible because, as [23] has already demonstrated, summary graph construction from a generic identity transform can be made completely without loss of information.

Only two good test cases were found. Both use the `xmlspec` schema on the input side, and `XHTML` cf. `XML-FO` as output.

About the legacy DTD-based version of xmlspec, [23] wrote they it had too many errors for practical evaluation. We largely agree: It is an old stylesheet, having had four different editors and numerous changes in its specified input language.

[61]In fact, no validation back-end existed for the first 3/5 of the project's duration

[62]hence the d'oh!

Category	count
`choose`-guarded attribute copying	21
Whitespace normalization bug	58
`xsi:type`, `xsi:nil`	30
Parameters taking default empty value	124
Invalid child	414
Required attribute missing	1
Invalid (empty) contents	2

Table 17: First impression on fraction of spurious errors in a large stylesheet which [23] gave up because of its (an older DTD-based version) many errors: There are still many errors in the stylesheet. Many real errors stem from a template that matches `*`, and emits highlighted warnings (in HTML) whenever no other template matches. Choose-guarded attribute copying is a `copy` instruction inside a `choose` copying some attributes — some, if not all of our errors from that are definitely spurious. "Whitespace normalization bug" are spurious errors caused by a missing whitespace normalization of some values from input — they are `concat`enated onto a constant, and afterwards may have internal whitespace, which the output schema does not accept.
The "parameters taking default value" category is caused by the flow-insensitive template parameter resolution: The relevant template seems to always be called with a nonempty parameter, but the default value is the empty string, causing the error. The stylesheet is an XSLT1 one; in XSLT2, this can be avoided.

It failed validation with an impressive 650 errors, which were partitioned roughly in Table 17; roughly one third are spurious; the rest are real[63]

Spurious errors here appear largely to be due to stylesheet authors having lost control, and using predicates for controlling flow at places where selection semantics had been more appropriate. We suspect that many of these predicates were added during frantic efforts to get the transformation results of single input instances to schema-validate!

Thirty errors were due to `xsi:type` and `xsi:nil` attributes with input-specified types being `xsl:copy`ed to the XHTML output, where they were undeclared.

The schema has a history dating back to 1998, with numerous revisions and additions since, including the third-party specifications included, and with an editor backlog of four people. This has lead to templates like this one:

```
<!- Silly HTML elements used for pasting stuff in; shouldn't ever
show up in a spec, but they're easy to handle and you just
never know.  ->
<xsl:template match="a|div|em|h1|h2|h3|h4|h5|h6|li|ol|pre|ul">
...
```

[63]One and a half days were spent debugging the validator after getting this result, looking for the bug that caused a set of official W3C releases to err that badly. The lesson learned was that (again) there *was* no bug: It the beginning of the validator's life as running code, maybe 80% of strange-looking results were due to bugs in the validator. Later, after code maturity, most surprising results were *really* validation errors: Test cases finding bugs in program the program code had become program code finding bugs in test cases...

This first experience with validating a large schema/transform/schema triple indicates that the number of errors grows out of proportion with the schema/stylesheet size, and that the size of the whole schema can grow towards loss of overview and control ("spaghetti") without proper power tooling.

The salesreport example

Despite a number of challenges to a validator (human or automatic), our example transform *validates*; it always produces valid XHTML, given valid sales reports.

Some challenges, with requirements for a validator not to spuriously err, are:

- There are two `sales`-matching templates, one absolute and one not. Both had correct priority computed; on the other hand, both were also used.
- Evaluation of the `concat` function in r_3, over an enumerated simple type and a string literal worked fine
- The `choose` instruction in t_3 retained precision
- There are two different templates r_5 and r_6 matching elements named `name`, with a risk that a buggy validator confuses flow to the two. That did not happen.
- The `width="{text()}"` AVT in the last template rule copied to an integer attribute from an integer datatype. No errors were reported.
- URL datatypes copied OK.

Conclusion

To the degree tested, the only two aspects in our static XSLT validation that still need improvement for a practical on-line application in an authoring tool are: Performance and sensitivity towards predicates.

Performance

The author of BRICS Schematools, Anders Møller, informed us that there is still room for large performance improvements ("factor 10") in the validation algorithm. With such an improvement, things would become much more useful.

Another possible improvement is to apply validation only to content models of elements generated by the transform summary graph that have recently changed because of edits: A difference graph between two flow graphs - a previous-generation one and a next-generation one — could be computed as an add-on to the flow analysis. Validation should then be applied only to the elements in the constructed summary graph that had their content model changed during the generation. This is entirely possible within BRICS Schematools, but not yet implemented.

Proposal: Predicates and drop-off decorators

Our general impression on precision of the flow analysis is that in most respects, it is excellent: *No* examples were found — and none successfully constructed — that made elements with the right *name* but wrong *DNT* become context types of some template in the analysis. The path extension trick seemed overall sufficient to keep flows separate, even in complex or poorly designed cases.

By far the most spurious errors, particularly in the XML Schema input- and output typed xmlspec test case, were due to *missing predicate tests.* The xmlspec case seemed, ironically, to be particularly bad because of its authors lacking static validation, driving them into (late-night?) patching it with predicates, at points where their *dynamic* validation failed[64].

Predicates are very hard to analyze statically and generally: Even Milo et. al., who aimed at a sound and complete analysis for a sublanguage, had to restrict away cases where data values were being joined (compared). Møller et al. suggest using “a primitive theorem prover”, but do not go on to pursue it any further. We bring a suggestion for a simple predicate checker — so simple, that it should be easy for users to identify the cases that it will recognize vs. the cases it will give up upon[65]:

The kinds of predicates recognized will be those that assert something within one `child`, `parent` or `attribute` step's distance of a node:

- `[attribute::foo]`, `[attribute::foo="`*value*`"]`
- `[child::foo]`
- `[parent::foo]`
- `[name(self::node())="`*n*`"]`
- any of the above, with a `not()` function around the expression

Predicate tests could now be designed to work rather much like node test ($S^{\cdots}_{test}(\Omega)$), except that where node test only *retained* or *discarded* DNTs, a predicate that may *retain*, *discard* or *redecorate* a DNT.

[64]This is pure speculation, though. We do not know who had authority to modify the transform and at what time. But the transform does bear traces of something like that, and surely could benefit enormously from static validity.

[65]The predicate in question could also just be highlighted in the editor, and a fly-over text could explain what the validator “thinks” it means

The decorator is a standard Decorator design pattern instance ([13]), that will modify some of the functions defined in Section 6.8 on the DNT. As a decorator, it will have all the properties of a DNT, some modifying, and additionally a special operation $\mathit{accept}(q, p)$, which determines if some DNT q can pass the predicate test p at all. If definitely negative, the DNT is discarded in the predicate test. Otherwise, the DNT is replaced by a decorator over it:

Example: $p =$ `[attribute::foo]`. The predicate is syntactically recognized as a case for an "attribute-present" decorator, found in a decorator library. The $\mathit{accept}(q, p)$ operation accepts a DNT if it has a declared attribute named `foo`. Once attached to the DNT and now itself being a pseudo-DNT, the decorator declares the `foo` attribute Required.

Example: $p =$ `[not(child::foo)]`. The predicate is syntactically recognized as a case for an "content-model-restrictor" decorator. The $\mathit{accept}(q, p)$ operation accepts a DNT if its content model expressed as a regular language, does not result in the empty language when intersected with $\mathcal{E}_d \backslash (e^i_{foo})^*$ (a language of all content models with no `foo` elements). Once attached to the DNT, the decorator removes all declared `foo` children from its content model ($C_d(q_{decorated}) = C_d(q) \backslash \{e^i_{foo}\}$ is all the flow analysis needs).

Example: $p =$`[name(self::node())=`n`]`. The predicate is syntactically recognized as a case for an "extra-name-test" decorator, its *accept* operation just being an extra name test, and the rest of it not modifying any properties of a DNT.

The technique will also apply for `test` expressions in XSLT's `when` elements.

In the flow analysis, decorated DNTs will flow around between templates, modeling the flow during a real transform more faithfully than ever. Some care must be exercised, though, when flows of some decorated DNT *merge* at a template with flows of the same DNT without a decorator, or with a different decorator: Conservativity could be endangered. One workable solution could be to consider

$$\{d(q)\} \sqsubseteq \{q\}$$

in lattices, such that a a re-propagation of a decorator-less DNT will happen automatically when it is added to a context set, and to enforce: Whenever two different decorators over the same DNT q appear in the same context set, both are removed and q is added (hence our term drop-off decorators).

16 Code-assist algorithms

We follow up on our experiments with the several smaller code-assist algorithms suggested. In general, neither is still integrated into the UI of the exploratory editor program, but the algorithm code exists and, and we have tested simply that neither generally returns so little data that it hardly contributes with anything interesting, and still does not return so much information that it would be of no help to the user.

16.1 Absurd patterns and selections

A pattern is ***absurd*** relative to a schema, if it never matches a node in any instance document that is valid with respect to the schema.

A path expression is ***absurd relative to a DNT*** q in a schema d if: For any instance document valid wrt. to d, the path expression always selects an empty sequence when the context node for the its evaluation is of type q.

A path expression is ***absolutely absurd*** with respect to a schema d if: For any instance document valid wrt. to d, for any DNT q in d: the path expression is absurd relative to q.

Absurd paths and expressions are interesting to know about in an XSLT authoring context, because:

- They reveal discrepancies between the way XML is used stylesheets vs. the way it was defined in schema, very often version discrepancies.
- They reveal flow leaks into code that is not intended to be used, like legacy code or built-in rules
- The exact way that XML namespaces are bound in XPath expression is a source of much confusion and time wasting: Unintended namespace use will almost certainly lead to absurd patterns or expressions, and can thus be caught without the need of debugging.

The flow analysis algorithm was instrumented with absurd expression detection code:

For absolute absurdity of `apply-templates` selections and template rule `match` patterns (call them both p), the test is simply

$$abs(p) = \begin{cases} \textbf{absurd} & \text{if} \qquad S(p, \Sigma_d) = \emptyset \\ \textbf{ok} & \text{otherwise} \end{cases}$$

For `apply-templates` instructions selections, the relative absurdity test is

$$abs(p, q) = \begin{cases} \textbf{absurd} & \text{if} \qquad S(p, \{q\}) = \emptyset \\ \textbf{ok} & \text{otherwise} \end{cases}$$

Relative absurdity in a selection does not imply absolute absurdity; there may be other context types. `value-of` expressions were not tested, but they are worthy candidates for such a test, too: 8 examples were found (during some different work) in the test set of the kind `<value-of select="e/text()"/>`, where the type `e` was declared without text content.

Results from the test set are:

The seven test cases not shown have no absurd patterns or selections at all.

Test case	n_t	n_a	n_r	comment
adressebog	1	0	0	No attributes declared; built-in attribute rule absurd
dsd2html	19	0	181	Inclusion of and later removal of schema part?
links2html	2	0	1	
ontopia2xtm	2	0	3	
order2fo	0	0	1	
purchaseorder	0	0	6	
window2xhtml	0	0	0	
xhtml2fo	1	0	2	
xmlspec	7	0	121	

Table 18: Relatively and absolutely absurd selection expressions. n_t is the number of template rules with absurd patterns. n_a is the number of absolutely absurd selection expressions in live code. n_r is the number (selection expression, DNT) pairs for which the DNT was a context type of the template containing an `apply-templates` instruction holding the selection expression, and the selection expression was found to always select an empty sequence. Test cases not shown have no template rules with absurd patterns, or absurd selections.

Even though there is not a single absolutely absurd selection, we still believe that such a test is valuable in an editing context, because it is very cheap complexity-wise, and it catches simple mistypings and namespace errors.

Conclusion

This test seems very suitable for generating on-line highlighting warnings that something is definitely wrong with an expression, as evaluated in the context of a schema.

16.2 Dead template detection

Needless to say, this analysis is quite simple: Is merely looks for empty (for any mode) context sets after flow analysis. If XSLT's `tests` in `if` and `choose` were attempted evaluated, their sequence constructors could also be added to dead code analysis, but stylesheet authors typically use these instructions exactly to construct something that is hard to argue about statically.

Some programmers consider it bad style to use built-in templates: [11] call that having a *missing* template. For those, this dead code analysis is useful to verify that all built-in templates remain unused.

Test case	n_u	n_b	comment
adressebog	0	1	The dead built-in rule is the `attribute::*` one
agenda2xhtml	0	4	
availablesupplies	0	6	
dsd2html	19	3	The absurd-pattern rules, and some built-in rules
emaillist	0	3	
links2html	2	2	
ontopia2xtm	2	2	
order2fo	2	3	
poem2xhtml	0	4	
purchaseorder	0	4	
slides2xhtml	0	1	
sqlprocedures	0	6	
staticanalysis	0	3	
window2xhtml	0	3	
xhtml2fo	4	1	
xmlspec	0	1	

Table 19: Counts of dead templates. n_u is the count of dead templates in the user stylesheet modules; n_b is the count of dead built-in templates.

Conclusion

Dead template detection is, of course, a really simple analysis. It caught almost no dead code in small stylesheets, which is hardly a surprise since these were probably just written, tested, deployed and not maintained later. The large xmlspec stylesheet has a revision history dating back to 1998, with a large fraction of legacy code; the newest, XSD-specified version had 36 dead templates.

16.3 Plausible, but *absent* flows

We want to present to the stylesheet author, given a template-invoking instruction and a context type, the reasons that some edge flows from the instruction when invoked with the context type, do *not* exist. Of course, it course, targets for *every* absent target template could be reported, but that would result in too many obvious, uninformative messages.

Recall $F_l \ : \ \mathcal{I} \rightarrow \mathcal{M} \rightarrow \mathcal{R} \rightarrow 2^{\Sigma_u}$

We expect that a stylesheet author looking at a stylesheet for a little while will not expect *more* flows to exist than what F_l returns — it *is* a rather rough approximation, and so look into using the set difference

$$miss(m, q, i, r) = F_l(i)(m)(r)\hat{\setminus}F(m)(q)(i)(r)$$

— where the $\hat{\setminus} : \Sigma_u \times \Sigma_d \rightarrow \Sigma_d$ operator expands e_*, a_* on its left-hand-side to $\mathcal{E}_d$ and $\mathcal{A}_d$ before doing normal set difference (a detail).

A few experiments showed this *miss* to return too many flows to be informative, when $select(i)$ was on one of the forms a`::node()` or a`::*` for any axis a, so the $\hat{\setminus}$ semantics was changed to effectively expand e_*, a_* on its left-hand side to $\emptyset$, ignoring any-type schemaless flows.

With some code instrumentation added in the abstract evaluation and template competition analyzes, the *cause* of a flow being removed can be queried:

1. The flow is infeasible under input schema validity (including path expression absurdity)
2. The flow is overridden by a template with a higher import precedence
3. The flow is overridden by a template with a higher priority

Further, death reporting of flows to built-in templates was kept separate from that of flows to user template rules, to enable filtering them away. Call a set of edge flows for some fixed i, r:

$\bigcup_{m \in \mathcal{M}, q \in \Sigma_d} miss(m, q, i, r)$

a ***bundle*** for the instruction i and template r; these are missing flows that can be presented together in an UI.

Some test cases appear to have dead code, but no flows to dead code have been removed. In fact such flows have been removed, by the template competition analysis stage of the schemaless flow analyzer, and does not appear in the table.

For each bundle except xmlspec, the number of bundles is quite moderate compared to the stylesheet line count, meaning that this absent flow information could easily be displayed in a UI, and that it is not too verbose to be informative.

Conclusion

A little bit of inspection of the data indicated that it is quite good; most reporting schema incompatibility and incompatible intermediate steps quite sharply. The ultimate evaluation of its utility will not come before a presentation of it is implemented in a UI, but it has been demonstrated that the amount of data to be presented is quite appropriate — there are several lines of space for each.

Test case	n_{bp}	n_{bep}	n_{bi}	n_{bei}	n_{ui}	n_{uei}	l
adressebog	2	2	0	0	0	0	76
agenda	1	1	0	0	0	0	43
availablesupplies	0	0	0	0	0	0	42
dsd2html	4	4	14	14	0	0	1353
emaillist	16	16	15	15	0	0	257
links2html	12	24	4	4	3	3	128
ontopia2xtm	8	9	8	8	2	2	188
order2fo	6	6	10	10	1	1	112
poem2xhtml	7	4	0	0	0	0	35
purchaseorder	5	5	4	4	0	0	112
slides2xhtml	11	26	6	6	0	0	118
sqlprocedures	14	14	9	9	0	0	258
staticanalysis	24	53	20	20	7	7	262
window2xhtml	14	43	13	13	0	0	701
xhtml2fo	27	845	5	5	3	3	1697
xmlspec	31	33	16	17	604	614	2528

Table 20:
n_{bp}: Number of nonempty bundles, where the target template is built-in. The cause of flow removal was import precedence override.
n_{bep}: Number of absent edge flows to built-in templates. The cause of flow removal was import precedence override.
n_{bi}: Number of bundles, where the target template is built-in. The cause of flow removal was schema incompatibility.
n_{bei}: Number of absent edge flows to built-in templates. The cause of flow removal was schema incompatibility.
n_{ui}: Like n_{bi}, but targeting user templates. The cause of flow removal was schema incompatibility.
n_{uei}: Like n_{bei}, but targeting user templates. The cause of flow removal was schema incompatibility.
l : Number of lines in the stylesheet.

16.4 Unused input analysis

This simple analysis can find all DNTs that never become context types during a transform, and never are selected in a `value-of` instruction, and for all declared children and attributes of which the same statement holds. Under some assumptions, a pre-transform can then be generated that strips subtrees rooted in any of these DNTs from XML input, before being fed to the main transform. In cases where large amounts of data is ignored (queries), this could substantially save time and memory.

The usage of a generated pre-transform is not always safe, even if it does not remove data actually accessed: Just consider an element declaration e^i_x with the content model

$\mathsf{Seq}(r, s, t)$

and an XPath expression selecting an `s` child an e^i_x context node as `child::*[2]`. It should be evident that if `r` children are removed in a pre-transform, the selection will now select something different than before.

The analysis works simply by saving into a set s_1:

- All DNTs that are found to be context types of some template
- All DNT that are in the result of abstractly evaluating any `select` expression on XSLT instructions in live code. For `value-of` instructions that evaluated to $\{$**pcdata**$\}$, the set of element DNTs that the semifinal step in the instruction's `select` expression was saved instead, or, if the length of the (path) expression was 1, the context type for the evaluation was saved. This was to avoid saving **pcdata** itself, which would later prevent *any* declared ancestor of **pcdata** from being filtered away.

and then

- Removing **comment** and `pi` from s_1. The safety of this move is under the (reasonable) assumption that the transform does not use upward steps from any of these types.
- computing a second set s_2 containing `ancestor-or-self` DNTs of all DNTs in the set, taking care not to consider only DNTs that were used as context types and data carries, but also the ancestor types that contain them.
- Finally computing $u = \mathcal{E}_d \backslash s_2$, this being the set of element types that neither are used directly themselves, or have declared descendants that are used.

Attributes were ignored, as they are hardly worth removing.

Test case	$\lvert u \rvert$	n_t	d	f
adressebog	0	11	0	24
agenda2xhtml	1	6	0	3
availablesupplies	2	18	0	2
dsd2html	0	40	0	4762
emaillist	1	20	0	28
links2html	2	12	0	15
ontopia2xtm	0	11	1	23
order2fo	10	19	0	11
poem2xhtml	0	7	0	5
purchaseorder	6	22	0	9
slides2xhtml	0	13	0	13
sqlprocedures	0	14	0	15
staticanalysis	0	22	0	39
window2xhtml	5	37	0	31
xhtml2fo	7	89	0	554
xmlspec	20	178	20	500

Table 21: Number of unused input element declarations and summary graph fragment counts. $\lvert u \rvert$ is the number of unused element declarations; n_t is the total number of element declarations in each input schema. The large count for order2fo is because of an absurd selection that misses the subtree rooted at the `ShipTo` element declaration.
d is the number of summary graph fragments generated which have no reachable outputting fragment, and f is the total number of fragments.

Conclusion

This algorithm is so inexpensive in computing resources that there was no need of measuring that. It found a fair number of unused element declarations, which could be used in a pre-filter generator, or as an information to stylesheet editors that there is still some input data that has been missed (where completeness is required).

16.5 No-output-reachable analysis

This algorithm test for subgraphs of the constructed summary graph that never generate output. It was observed that some stylesheets processed some subtrees of their input recursively, but seemed never to output anything in these traversals.

The summary graph fragment generated for each ($m \in \mathcal{M}, q \in \Sigma_d, r \in \mathcal{R}$) was marked, depending on whether or not r had any `element`, `attribute`, `copy` or `value-of` instructions. Then, summary graph edges between all template fragments were (conceptually) reversed, and the closure of all fragments with a reachable output instruction was computed. Finally, the edges were reversed back to normal.

Conclusion

This analysis could actually catch some cases of time-wasting traversal of unused subtrees of the input, provided that our (upper-approximate) flow analysis could prove that no flow would go to templates with output instructions. However, this was hardly ever the case; maybe because of the built-in rule for **pcdata**, that outputs copies of text nodes.

Part V

Conclusion

17 Some results

We have pushed the envelope of static analysis yet a little bit further:

- A method for fast and quite precise XSLT2 flow analysis under XSL Schema validity of input has been found.
- The time and memory performance of general XSLT flow analysis has been improved enormously; this of course extends to XSLT1 and to other programs where XPath

pattern matching is used.

- Static XSLT validation had been extended to cover XSLT2 and XML Schema.
- It has been demonstrated that XSLT flow analysis is now fast enough for real-time code-assist use, even for large stylesheets and large schema.
- The claims above have been verified using an existing test set.
- XSLT2 with its partially typed transforms is still not a W3C Recommendation, and it is too soon for full-scale empirical tests.
- However, test cases were constructed for most apparently difficult cases, and no major problems were found.

A performance bottleneck in the best (known) existing practical XSLT flow algorithm that prohibited on-line XSLT flow analysis for large instances, was eliminated without loss of precision. However, it turned out that validation was too slow for large instances!

This needs not be definitive: The summary graph validation in BRICS Schematools could be altered to validate only a selected set of nodes, and an algorithm could then be made that computes the difference between generational flow graphs. This is probably the first problem to be solved, before on-line static *validation* (not just flow analysis) becomes practical for all realistic instance sizes.

All proposed code-assist algorithms were implemented and evaluated for usability on real stylesheets, and all seem to be able to provide some real, practical help. This will ultimately show when they are integrated into the editor with proper UI visualization.

The final test with the XML Schema input-defined xmlspec stylesheet showed that precision still should be improved for practical use: Predicate evaluation and a context-sensitive template parameter resolution are necessary to get the error count for this case down to where some confidence can be established for the user that he has real errors when the tool tells him he has errors. Alternatively, some way should be found to identify the flow through a stylesheet between the bad construct (if there is one) and the element that contains it — like decorating summary graph edges with origin information, or through a no-semantics information node type.

18 Demonstrator application

An XSLT editor application was made. Its back-end code can be hooked up with just about any front-end, though — like the one at `www.dongfang.dk/xslv/xslv.html`

18.1 README — download and build instructions

The program can be retrieved from

`http://www.dongfang.dk/xslv/xslv.tar.gz`

It requires Java 1.5, Apache Ant and BRICS Schematools. The archive should be unpacked so that its root is a sibling of the **schematools** root (otherwise, the `summarygraph.home` property in **build.xml** can be changed to point to the actual **schematools** directory).

Schematools needs a slight addition: In **src/dk/brics/relaxng/converter/RNGParser.java**, add

```
/**
 * Parses schema from InputSource.
 * @param source - an InputSource
 * @return schema
 * @throws ParseException if error occurs during parsing
 */
public Grammar parse(InputSource source) throws ParseException {
  try {
    builder.setInitialOrigin(new Origin(source.getSystemId(), 0, 0));
    return parse(builder.build(source), new URL(source.getSystemId()));
  } catch (JDOMException e) {
    throw new ParseException(e);
  } catch (IOException e) {
    throw new ParseException(e);
  }
}
```

Then, rebuild Schematools.

After unpacking the first time, the included third-party libraries need to be unpacked and built. This is done by changing to the **speciale/lib** directory, and starting Apache Ant (`ant`).

To build the XSL tool kit, issue `ant` from the **speciale** directory.

18.2 Program Manual

Validation — command line

To validate a single instance from the command line, use `java dongfang.xsltools.validation.XSLTValidatorMain` *stylesheet input−schema output−schema* [*input − namespace*] [*input − root*]

stylesheet, *input − schema* and *output − schema* may be given as absolute or relative file names, or as URLs. *input − namespace* and *inputroot* are the namespace (for DTD) and the root element name (for DTD and XML Schema) for input. The input schema language is auto-detected to DTD or XML Schema, but only reduced Relax NG schemas can currently (until the BRICS Schematools XML Schema front end is finished) be used as output schema language.

input − namespace may be omitted; in that case the validator defaults to no namespace.

input − root may be omitted; the validator will auto-guess in that case, and print the guess (which should be verified).

The Java class path is assumed to be configured (everything needed is included, or comes with Schematools):

- the **build** directory
- **dom4j.homebrew.jar** in a **build** directory of the **lib/dom4j** directory
- **dtdparser.jar** in **lib/dtdparser**
- the **build** directory of BRICS Schematools
- all **jar**s in the **lib** directory of BRICS Schematools
- **schematools.jar**s in the **dist** directory of BRICS Schematools

Validation — Apache Ant

This has the advantage that the classpath is pre-configured.

`ant validate -Dstylesheet=`*stylesheet* `-Dinput-schema=`*input−schema* `-Doutput-schema=`*output−schema* [`-Dinput-namespace=`*input − namespace*] [`-Dinputroot=`*inputroot*]

The meaning of the parameters and the results are the same as for command line validation.

Triple files

To support bundling of stylesheet with schema resources and contextual (namespace, root element name) information, a test-triple XML format was designed. The format is, by prototype:

`<triple name="`*name*`" enabled=`*enabled*`>  <input type="`*inputlanguage*`" DTDNamespaceURI="`*inputnamesp*
root`">`*input−schema*`</input>  <output type="`*output−language*`">`*output−schema*`</output>`
`<xslt>`*stylesheet*`</xslt> </triple>`

input − language and *output − language* may be safely ignored. The other parameters are as explained above.

Running triple files — Apache Ant

`ant runtriple -Dtriple=`*triple* [`-Duse-automaton-algorithm=use-automaton-algorithm`] [`-Donly-xcfg=only-xcfg`] [`-Donly-sg=only-sg`] [`-Donly-test-files=only-test-files`] [`-Dnum-runs-each=`*n*] [`-Dwait-each-triple=wait-each-triple`]

triple is the a triple file name. If the file is a directory, all its *sub*directories are searched for **triple.xml** files, and all such triples are run, if enabled.

`use-fast-algorithm` disables the context-sensitive ancestor language analysis, `only-xcfg` terminates the validation process when the control flow graph has been constructed, `only-test-files` terminates the process when all files referred to in the triple have been loaded, `num-runs-each` specifies the number of times to run each triple and `wait-each-triple` waits for keyboard feedback after each run.

Deploying the interactive web validator

This is basically a done by installing Tomcat, and editing the `tomcat.home`, `webapp.name`, `java.handler.extension` and `web.classpath` properties in **build.xml**. The handler extension is needed to make the URL resolution internally in the validator use the web service interactively for resolving resources (using a custom protocol in URLs, **dongfang**).

dk.brics.misc.Automata needs to be altered to load automata over the class loader of the `Automata` class, *not* the system class loader. Otherwise, automata will not load under Tomcat's modified class loaders.

`ant deploy` will copy everything needed to the servlet container — and also a custom protocol handler to the **ext** directory of the JVM. Permissions probably need to be set before this will work.

Theoretically, the web application should work then. Tomcat and its JVM needs to be restarted, with the JVM parameter `-Djava.protocol.handler.pkgs=dongfang.xsltools.resolver`.

It was only ever tested on one location, so deployment can be expected to take a little experimentation before success.

Running the interactive editor

`ant editor -Dtriple=` *triple*

There is currently no save or load capability in the editor; it will only start from a triple. It could be trivially added, but for the present feasibility study, it is not necessary (and the UI thread stays running even if the validator thread crashes, leaving the user a chance to grab his mouse, and copy out his data).

18.3 Configuration Parameters

Some configuration parameters are available, allowing control of:

- Control flow algorithms used
- Data logging and dumping
- Sanity tests used

Configuration is not dynamic, i.e. it requires modification of a Java source file, and a rebuild:

Class dongfang.xsltools.controlflow.ControlFlowConfiguration

- `boolean killCandidateEdgesOnlySoftly()`: Returning **true**, the ... tests will not actually kill candidate edges (as discussed in...), but will only mark them as potentially removed. Used for sanity checks.
- `boolean useColoredContextTypes()`: Enables or disables colored context types, as discussed in ...
- `boolean useColoredFlowPropagation`: Enables or disables re-propagation of context flow for each color of a context type. If disabled, flow propagation will only take place the first time a context type is added to each template rule context set. If enabled, context flows will be re-propagated for each new *color* of each context type of each template rule.
- `boolean useCommentPIPropagation`: Enables or disables generation of summary graph fragments for **comment** and **pi** context types. For stylesheets in which some nonempty templates that match either of those types, this property should be **true**. However, such stylesheets are very rare, and for all other stylesheets, **comment/pi** summary graph fragments will be empty, and will just clutter up the summary graph. Setting this property to **false** will eliminate them.
-

Class dongfang.xsltools.controlflow.ModelConfiguration

The following two validation features work on XML representations of stylesheet modules or schema passed to the static analysis algorithm. They are not related to what we term (static)"XSL validation", but simply serve to enforce consistency in the data passed to our algorithms, preventing some kinds of spurious errors.

- `boolean schemaValidateXSLTSource()`: Whether to validate the XSLT2 stylesheet modules, using an XML Schema for XSLT2.0.
- `boolean schemaValidateXMLSchema()`: Whether to validate XML Schema input schemas, using the W3C Schema for Schemas.
- `boolean continueAfterXSLTParseOrValidationError()`: Whether the validator should ignore parse or XML validation errors in input (stylesheet / schema) parse phase, and try to continue.

Class dongfang.xsltools.simplification.SimplifierConfiguration

- `boolean shouldContinueStylesheetProcessingAfterSyntacticErrors()`: Whether the validator should ignore semantic errors encountered during simplification phase, and try to continue.
- `boolean removeVariableDeclarations()`: Whether the simplifier should remove variable declarations from the stylesheet.
- `boolean removeAttributeSetDeclarations()`: Whether the simplifier should remove attribute set declarations from the stylesheet.
- `boolean removeParameterDeclarations()`: Whether the simplifier should remove parameter declarations from the stylesheet.
- `boolean dumpIntermediateResultsAtEachStage()`: Whether the simplifier should save a copy the stylesheet primary module prior to and posterior to each stage of the simplification, in the directory named by the `String intermediateDumpPath()` method. Useful for diagnostics, but affects performance.
- `WhitespaceNodeBehaviour whitespaceNodeBehaviour()`: Whether the simplifier should preserve or remove whitespace-only text nodes outside of `xsl:text` nodes in the stylesheet.

References

[1] XML schema part 0: Primer second edition, 2004.

[2] XML schema part 1: Structures (second edition), 2004.

[3] XML schema part 2: Datatypes (second edition), 2004.

[4] RELAX NG specification, 2001.

[5] Extensible Markup language (XML) 1.0 (Third Edition), 2004.

[6] XML Path Language (XPath), 1999.

[7] XML Information Set (Second Edition), 2004.

[8] Xquery 1.0 and xpath 2.0 data model(xdm), 2005.

[9] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A formal model for an expressive fragment of XSLT. *Inf. Syst.*, 27(1):21–39, 2002.

[10] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *WWW*, pages 712–721, 2005.

[11] Ce Dong and James Bailey. Static analysis of xslt programs. In *CRPIT '04: Proceedings of the fifteenth conference on Australasian database*, pages 151–160, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[12] Anders Berglund et al. XML path language (xpath) 2.0. http://www.w3.org/TR/xpath20/, 2005.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, March 1995.

[14] Koshuge Kawaguchi. W3c XML schema made simple, 2001.

[15] Michael Kay. XSL transformations (XSLT) version 2.0. http://www.w3.org/TR/xslt20/, 2005.

[16] Stephan Kepser. A simple proof for the turing-completeness of XSLT and XQuery. In *Extreme Markup Languages*, 2004.

[17] Ashok Malhotra, Jim Melton, and Norman Walsh. Xquery 1.0 and xpath 2.0 functions and operators(cr), 2005.

[18] Wim Martens, Frank Neven, and Thomas Schwentick. Which XML Schemas Admit 1-Pass Preorder Typing? *ICDT*, LNCS(3363):68–82, 2005.

[19] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, 2000.

[20] Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, October 2005.

[21] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[22] Frank Neven. Automata theory for XML researchers. *SIGMOD Rec.*, 31(3):39–46, 2002.

[23] Mads Østerby Olesen. Static Validation of XSLT. Master's thesis, University of Aarhus, 2004.

[24] Michael I. Schwartzbach. Lecture notes on static analysis. University of Aarhus, 2005.

[25] Akihiko Tozawa. Towards static type checking for XSLT. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document engineering*, pages 18–27, New York, NY, USA, 2001. ACM Press.

[26] P. Wadler. A formal semantics of patterns in XSLT, 1999.

[27] Baltasar Trancon y Widemann, Markus Lepper, and Jacob Wieland. Automatic construction of XML-based tools seen as meta-programming. *Automated Software Engineering*, 10(1):23–38, 2003.