BRICS RS-05-32 Møller et al.: Static Validation of XSL Transformations

BRICS

Basic Research in Computer Science

Static Validation of XSL Transformations

Anders Møller
Mads Østerby Olesen
Michael I. Schwartzbach

BRICS Report Series RS-05-32

ISSN 0909-0878 October 2005

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/05/32/`

Static Validation of XSL Transformations

Anders Møller[*], Mads Østerby Olesen, and Michael I. Schwartzbach

BRICS[†], Department of Computer Science
University of Aarhus, Denmark
{amoeller,madman,mis}@brics.dk

October 28, 2005

Abstract

XSL Transformations (XSLT) is a programming language for defining transformations between XML languages. The structure of these languages is formally described by schemas, for example using DTD, which allows individual documents to be validated. However, existing XSLT tools offer no static guarantees that, under the assumption that the input is valid relative to the input schema, the output of the transformation is valid relative to the output schema.

We present a validation technique for XSLT based on the summary graph formalism introduced in the static analysis of JWIG Web services. Being able to provide static guarantees, we can detect a large class of errors in an XSLT stylesheet at the time it is written instead of later when it has been deployed, and thereby provide benefits similar to those of static type checkers for modern programming languages.

Our analysis takes a pragmatic approach that focuses its precision on the essential language features but still handles the entire XSLT 1.0 language. We evaluate the analysis precision on a range of real stylesheets and demonstrate how it may be useful in practice.

1 Introduction

XSL Transformations (XSLT) 1.0 [12] is a popular language for defining transformations of XML documents. It is a declarative language based on notions of pattern matching and template instantiation and has an XML syntax itself. Although designed primarily for hypertext stylesheet applications, it is more widely used for simple database operations or other transformations that do not require a full general-purpose programming language.

[*]Supported by the Carlsberg Foundation contract number 04-0080.

[†]Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

The term *stylesheet* is commonly used for a transformation specified in XSLT. Generally, a stylesheet transforms from one class of XML documents to another. The syntax of such a class of documents is specified formally by a *schema* using a schema language, such as DTD [9], XML Schema [35], or DSD2 [29]. An XML document is *valid* relative to a given schema if all the syntactic requirements specified by the schema are satisfied.

With XSLT being a specialized programming language it is natural to view the schemas as *types*. The notion of types is normally used in programming for detecting programming errors at an early stage in the form of type checking—however, the semantics of XSLT is independent of schemas. Our main goal is to remedy this through a mechanism for statically checking that the output of a given stylesheet is guaranteed to be valid relative to an output schema, under the assumption that the input is valid relative to an input schema.

Although the basic principles in XSLT are straightforward, it contains many features that cause intricate interplays and make static validation difficult. In fact, XSLT is Turing complete [21], so a fully automatic solution that is both sound and complete is not possible: the static validation problem is mathematically undecidable. We aim for a solution that conservatively approximates the behavior of the given stylesheet but has a sufficiently high precision and performance on typical stylesheets to be practically useful.

Example

Consider the following XML document which describes a collection of people that are registered for an event:

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

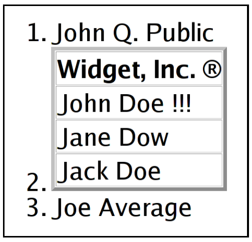
People are either registered as individuals or as belonging to a group. Each person has a unique `id` attribute. A `group` element has attributes identifying its `type` and its `leader` (referring to the `id` attributes). Such documents are described by the following DTD schema:

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

XSLT may be used to transform such documents into XHTML for presentation purposes. (A brief overview of DTD and XSLT is provided in Section 3.) We consider the following stylesheet:

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

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

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

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

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

Registrations are displayed in an ordered list, where people belonging to the same group are collected in a table. The affiliation of a private group is adorned with a ® symbol, and group leaders are indicated by triple exclamation marks. For the above example document, the resulting XHTML document is rendered as follows by a standard browser:



The question that we address is, for this example, the following: given an input document that is valid according to the above schema, will the stylesheet always produce an output document that is valid XHTML?

Contributions

The main contribution of this paper is an algorithm for statically checking validity of XSLT transformations, where we use DTD as the schema language for specification of input and output types. The algorithm is based on static analysis. It is sound in the sense that all validity errors are guaranteed to be detected, but incomplete since it may produce spurious warnings. If potential validity errors are detected, precise warning messages are automatically generated, which aids the programmer in debugging. To be able to design a precise analysis, we have investigated a large number of existing stylesheets resulting in some statistics about the typical use of the various language features.

Additionally, our algorithm is able to detect `select` expressions that never hit anything and template rules that are never used. These are not necessarily errors in the stylesheet, but presumably unintended by the programmer.

In a preliminary phase in our analysis, we simplify the given stylesheet into a *Reduced XSLT* language. This simplification involves different levels: some reductions are semantics preserving while others involve conservative approximation. We believe that such a simplification phase may also be useful in making other XSLT tools easier to design and implement.

Another central constituent of our analysis is an XSLT flow analysis that determines the possible outcome of pattern matching operations. This information may also be useful for other purposes than static validation, for example in XSLT processors for improving runtime performance or in XSLT editors for stylesheet development.

Moreover, we define a notion of *summary graphs* as a variation of earlier definitions, tailored to reasoning about XSLT stylesheets.

It is not a goal of this paper formally to prove soundness of our analysis, nor to discuss theoretical complexities of the algorithms we propose—instead we rely on informal arguments and practical experiments.

Overview

Our analysis builds on earlier results on static validation of XML transformations in the XACT project and its predecessors [23, 11, 10, 7].

We first, in Section 2, describe related work on analysis of XSLT. Section 3 provides a brief overview of DTD, XPath, and XSLT, and introduces the terminology that we use. In Section 4, we summarize results of our statistical investigation of a large number of existing stylesheets.

Section 5 presents the structure of our validation algorithm. The sections that follow describe each phase in detail. First, in Section 6, we simplify the given stylesheet to use only the core features of XSLT. In Section 7, the simplified stylesheet is subjected to a flow analysis which uses a fixed point algorithm to compute a sound approximation of the flow of template invocations. From this information together with the DTD for the transformation input, we are in Sections 8, 9, and 10 in a position to construct a summary graph, which is a structure that represents the possible outcomes of transformations using that particular stylesheet and input schema. Finally, in Section 11, we explain how this summary graph is validated relative to the output schema using a previously published algorithm [11].

In Section 12, we describe our prototype implementation and the results of applying it to a number of benchmarks.

2 Related Work

To our knowledge, no others have presented a solution to the problem of static validation for the complete XSLT language, although there are noteworthy results for fragments of XSLT and for other XML transformation languages.

An early attempt at static validation of XSLT is [3], which uses a set of typing rules to establish relationships between the input and output languages of XSLT transformations. Their goals are ambitious, but their method is only applicable to a tiny fragment of XSLT. However, the paper was influential in clearly defining the static validation problem.

The paper [36] examines a fragment of XSLT, called XSLT0, which covers the structural recursion core of XSLT. It uses inverse type inference, in the style of [28], to perform exact static output validation in exponential time. However, since XSLT0 only allows simple child axis steps in the recursion and ignores attributes, a reduction from XSLT to XSLT0 is only possible for the simplest transformations. Furthermore, the practical usability of the technique has not been demonstrated.

The work in [14] has the same aims as Section 7 of this paper: conservatively analyzing the flow of an XSLT stylesheet. Compared to our analysis, theirs is less precise in exploiting the information present in DTD schemas and XPath expressions. Also, [14] uses the control-flow information to detect unreachable templates and guarantee termination whereas we focus on the validity problem.

The article [31] presents a stylesheet that transforms XSLT stylesheets into

SVG representations of the possible control flow for the purpose of documentation and debugging. However, the precision is rather weak compared to [14] and our Section 7.

Continuing with the XSLT technology, numerous tools, such as [32, 1, 34], enable step-by-step debugging of stylesheets. Such tools are particularly useful during the development phase, but they cannot provide static validation guarantees. In fact, the popularity of XSLT debuggers seems to emphasize the need for tools as the one we provide here.

The correctness of our work depends on the formal semantics of XSLT, which is discussed in [37, 5]. Also, the algorithm in Section 7.4 is related to the work on analysis of XPath expressions presented in [38, 33, 4], though our problem in somewhat different and we sacrifice exact decidability for a useful conservative approximation.

The problem of static validation has been solved for more restricted formalisms, such as tree transducers. The work in [28] introduces the technique of inverse type inference to compute the allowed input language for a so-called k-pebble transducer given its output language. The resulting algorithm has non-elementary complexity. The paper [27] investigates how the expressive power of tree transducers must be further restricted in order to allow a polynomial time decision algorithm.

Static validation has been investigated for a host of other XML transformation languages, many of which have been designed with this feature in mind. Notable examples are XDuce [18], XQuery [15], XACT [23], XJ [16], and Cω [6] These languages cannot solve our problem, since neither supports an embedding of XSLT stylesheets that allows static validation of the resulting programs. A more comprehensive survey of this area is available in [30].

Our translation into Reduced XSLT in Section 6 may prove useful for other projects working on XSLT. For example, XSLT compilers such as [2] might benefit from translating only a smaller subset.

3 Background

We assume that the reader is familiar with XSLT and DTD, but to explain the terminology that we use, we recapitulate the main points in these languages and in XPath, which is an integral part of XSLT.

3.1 Document Type Definitions (DTD)

The DTD formalism is a simple schema language for XML and is described in the XML specification [9]. A DTD schema is a grammar for a class of XML documents defining for each element the required and permitted child elements and attributes. The *content* of an element is the sequence of its immediate children. It is specified using a restricted form of regular expression over element names and `#PCDATA`, which refers to arbitrary character data. Attributes can be declared as required or optional for a given element, and their values can be constrained to finite collections of fixed strings or to various predefined regular language of identifiers (such as `NMTOKEN`). An XML document is *valid* according to a given DTD schema if it describes the contents and attributes of all elements. Implicitly, we only consider well-formed XML documents, and we assume that entity references have been expanded.

When D is a DTD schema, we will assume that a specific root element, $root(D)$, has been designated (the DTD formalism does not by itself do this). Correspondingly, we use the notation $\mathcal{L}(D)$ to denote the set of XML documents with that root element name that are valid according to D. Thus, DTD schemas are similar to programming language types which also describe sets of allowed values.

Consider now our earlier example of a DTD schema:

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

The designated root element name is in this case `registrations`, whose content is defined to consist of an arbitrary sequence of `name` and `group` elements. The content of a `name` element is just a character data node and it has a single mandatory attribute named `id` of type `ID`. A `group` element must contain a a single `affiliation` element followed by any number of `name` elements and it has two mandatory attributes. The first, `type`, can only have the value `private` or `government`, whereas the second, `leader`, is of type `IDREF`. The `affiliation` node has as content a character data node and it has no attributes.

The attributes of type `ID` are required to have unique values throughout the document and, dually, those of type `IDREF` are required to have the same value as some `ID` attribute. This relationship is beyond the scope of our static validator, and, as other XML transformation type checkers [30], we ignore these attribute types in this paper.

3.2 XML Path Language (XPath)

XPath [13] is a simple but versatile notation for addressing parts of XML documents. It imposes a particular data model in which elements, attribute values, and character data are represented as *nodes* in a tree. The content of an element node is formed by a sequence of element nodes and character data nodes. It is not allowed to have two character data nodes as siblings in a tree. Attribute nodes are associated with a given element node as an unordered set.

An XPath *expression* can, relatively to an evaluation context, evaluate to a boolean, a number, a string, or a set of nodes. A node set expression is called a *location path* and consists of a sequence of *location steps*, each having three parts: (1) an *axis*, for example `child` or `following-sibling`, which selects a set of nodes relative to the context node, (2) a *node test*, which filters the selected nodes by considering their type or name, and (3) a number of *predicates*, which are boolean expressions that perform a further, potentially more complex, filtration. Thus, the result of evaluating a location step on a specific node is a set of nodes. A whole location path is evaluated compositionally left-to-right. A location path starting with `/` is evaluated relative to the root node, independently of the initial evaluation context.

Consider the XML documents described by the above DTD schema of which the earlier tiny XML document is an example:

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

On such documents, the XPath location path

```
//group[@type='private']/name[@id=../@leader]/text()
```

will select the names of the leaders of private groups, in this case `John Doe`. This example uses an abbreviated syntax which expands into the following expression in which all axis steps are made explicit:

```
/descendant-or-self::group[attribute::type='private']/
child::name[attribute::id=parent::node()/attribute::leader]/
child::text()
```

In the following, we use the notation $x \overset{p}{\rightsquigarrow} y$ to mean that evaluation of a XPath location path p starting at the node x in some XML document $X \in \mathcal{L}(D)$ results in a node set containing the node y.

3.3 XSL Transformations (XSLT)

XSLT, or XSL Transformations [12], is a declarative language for programming transformations on XML documents. It uses XPath as a powerful sublanguage for locating document fragments, performing pattern matching, expressing branch conditions, and computing simple values.

Consider the example stylesheet in Section 1. It has the following overall structure, which declares namespaces for the XSLT language itself (the prefix `xsl`), the input language (`reg`), and the output language (the default namespace):

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

```
                 xmlns="http://www.w3.org/1999/xhtml">
...
</xsl:stylesheet>
```

The remaining content of the stylesheet is a collection of template rules, such as this one:

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

The template rule has a `match` attribute, which defines the kind of nodes on which it may be applied. The value of this attribute is a location path (restricted to downward axes) and a given node is matched if it is a possible target of the location path (starting evaluation from *some* node in the tree). The body of the template rule is an expression, called a template, that evaluates to a fragment of the output document. As the above example shows, this is a mixture of literal fragments and computations (identified by the `xsl` namespace prefix). Another template rule shows a variety of such computations:

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

The `value-of` instruction evaluates the XPath expression given by the `select` attribute and converts the result into a string. The `if` instruction is a conditional that similarly converts the value of the `test` attribute into a boolean. The `apply-templates` instruction performs recursive invocations by using the `select` attribute to compute a sequence of nodes that must subsequently be processed.

XSLT is a large language, which we here consider in its entirety. In the remainder of paper we must deal with many features and subtleties that are described in detail in the specification [12].

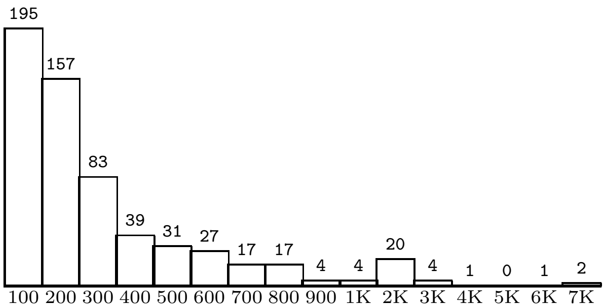


Figure 1: Sizes of stylesheets used in mining.

An XSLT stylesheet S may be viewed as a (potentially parameterized) map from XML documents to XML documents. We say that S is *valid* relative to the schemas D_{in} and D_{out} iff $\forall X \in \mathcal{L}(D_{in}) : S(X) \in \mathcal{L}(D_{out})$ (for *any* values of the stylesheet parameters). The challenge of *static validation* is to decide this property given S, D_{in}, and D_{out}, conservatively but with reasonable precision and efficiency.

Note that a stylesheet may use both *variables* and *parameters* that may each be *local* (defined inside a template) or *global* (defined at the top-level of the stylesheet). Each of these four cases will be treated differently in our analysis.

4 Stylesheet Mining

XSLT is a complex language with many peculiar features. To better understand how those are used in realistic applications, we have collected and analyzed XSLT samples. Googling for the XSLT namespace string, we have obtained 603 stylesheets with a total of 187,015 lines of code written by hundreds of different authors. These samples have then been subjected to various statistical investigations, the results of which have been remarkably stable once the collection went beyond a few hundred stylesheets. The sizes of the stylesheets measured in number of lines are distributed as shown in Figure 1. It indicates that most stylesheets are of moderate size, but a few are quite large. We have anecdotal evidence that some stylesheets are in the 100K range, but none so large has been available to us.

We are particularly interested in the complexities of XPath expressions used in `select` and `match` attributes. The samples contained 10,768 `select` expressions that can be divided into disjoint categories indicated by typical examples or brief descriptions as shown in Figure 2. The category *name(s) known* means that the name of the selected node is known to belong to a (small) set of constant node names. The “nasty” expressions, which resist reasonable analysis, almost all involve the `key` or `id` function or extension functions that are specific to a given XSLT implementation.

The samples similarly contained 8,739 `match` expressions which are broken

Select Category	Number	Fraction
default	3,418	31.7%
`a`	3,349	31.1%
`a/b/c`	1,173	10.9%
`*`	749	7.0%
`a | b | c`	480	4.5%
`text()`	235	2.2%
`a[...]`	223	2.1%
`/a/b/c`	110	1.0%
`a[...]/b[...]/c[...]`	101	0.9%
`@a`	68	0.6%
`/a[...]/b[...]/c[...]`	43	0.4%
`..`	34	0.3%
`/`	8	0.1%
name(s) known	602	5.6%
nasty	175	1.6%
Total	10,768	100.0%

Figure 2: Classification of `select` expressions.

down as shown in Figure 3. Here, the nasty expressions are those where the matched node is only characterized by a predicate.

An important conclusion from this statistical analysis is that the downward axes (`child`, `attribute`, `descendant`, `descendant-or-self`, and `self`) are dominant for `select` expressions with 92.5% (when we ignore XPath expressions that occur as predicates). Also, even when other axes are employed, it is usually fairly simple to determine *some* characteristic information about its target that allows us to limit its possible names to a small set. These observations form the basis for the approximation algorithm that we introduce in Section 7.4.

In later sections, we will mention other interesting observations that we have made on this collection of stylesheets.

5 Structure of the Validation Algorithm

Our analysis technique is inspired by the program analyses developed for the languages `<bigwig>` [8], JWIG [11], and XACT [23]. The `<bigwig>` language uses a notion of templates for constructing HTML or XHTML pages in interactive Web services, JWIG is a Java-based variant, and XACT generalizes the ideas to encompass general XML transformations. Using a lattice structure of *summary graphs*, originally introduced in the paper [7], the program analyses are able to provide static guarantees of validity of the output of programs written in these languages. The present analysis is also based on summary graphs, although we use a variant that is tailored towards analysis of XSLT stylesheets. (Compared to the earlier analyses, the one for XACT bears the closest resem-

Match Category	Number	Fraction
`a`	4,710	53.9%
absent	1,369	15.7%
`a/b`	523	6.0%
`a[@b='...']`	467	5.3%
`a/b/c`	423	4.8%
`/`	256	2.9%
`*`	217	2.5%
`a | b | c`	177	2.0%
`text()`	52	0.6%
`@a`	24	0.3%
`@*`	16	0.2%
`n:*`	12	0.1%
`processing-instruction()`	11	0.1%
`@n:*`	4	0.0%
`a[...]`	225	2.6%
`.../a[...]`	225	2.6%
`.../a`	108	1.2%
`.../@a`	24	2.7%
`.../text()`	11	0.1%
`.../n:*`	1	0.0%
nasty	97	1.1%
Total	8,739	100.0%

Figure 3: Classification of `match` expressions.

blance.) In Section 8, we formally define the notion of summary graphs that we use here. All we need at this stage is that a summary graph SG is a finite structure that represents a set of XML documents $\mathcal{L}(SG)$.

Given an input schema D_{in}, an XSLT stylesheet S, and an output schema D_{out}, we wish to construct a summary graph SG such that $S(\mathcal{L}(D_{in})) \subseteq \mathcal{L}(SG)$, that is, SG represents a conservative approximation of the possible output of transformations with S using input from D_{in}. We then check that $\mathcal{L}(SG) \subseteq \mathcal{L}(D_{out})$, that is, the transformation output is always valid relative to D_{out}.

We aim to construct SG such that $\mathcal{L}(SG)$ is as small as possible to avoid too many spurious warnings and such that the entire algorithm is efficient enough to be practically useful. Our approach is pragmatic. We aim to handle the full language, not just a toy subset. This requires us to focus the analysis precision on the essential language features, applying different degrees of approximation.

6 Stylesheet Simplification

The first phase of our approach simplifies the given stylesheet to use only a small number of core XSLT features. We divide the steps into two categories: some are semantics preserving, others introduce approximation. This simplification phase is quite complicated because of the intricate details of the many language features and their interplay. We here present highlights of the simplification steps and describe the resulting language *Reduced XSLT*.

We first, however, need to deal with the fact that XSLT contains a few special language constructs that are inordinately difficult to model with reasonable precision:

- We do not support the `text` output method; nor do we allow uses of `disable-output-escaping`. (If the output method is set to `html`, we automatically convert to XHTML and use XML mode.)
- We do not support any implementation-specific extension elements or functions, except a few ones introduced in the simplification.
- We ignore namespace nodes that are selected by `for-each` instructions or assigned to variables or parameters.

Whenever these constructs are encountered, a warning is issued. Unless the output method `text` is used, the analysis continues after applying a suitable patch, such as, replacing `value-of` instructions that use `disable-output-escaping` by elements with computed unknown names.

These limitations are not severe. Naturally, the `text` output method should not be used when XML is output. Also, according to the spec [12], "since disabling output escaping may not work with all XSLT processors and can result in XML that is not well-formed, it should be used only when there is no alternative". Namespace nodes are rarely selected explicitly in typical stylesheets. (In the 187,015 lines of XSLT mentioned in Section 4, it occurs only 6 times.) The typical use is in generic stylesheets that output a tree view of the input document where the selection of namespace nodes is not essential for the structure of the output documents.

To simplify the presentation, we assume that the input and output documents each use only one namespace. Our techniques, however, can be extended straightforwardly to accommodate multiple namespaces.

6.1 Semantics Preserving Simplifications

The simplification steps in the first category are semantics preserving, so they can in principle be applied in an initial phase of any tool that analyzes XSLT stylesheets.

First, we fill in defaults. The built-in template rules are inserted using explicit `priority` attributes to ensure that they have the lowest matching precedence. For each template rule without a `priority` attribute, the default priority is computed and inserted explicitly. Also, for each `apply-templates` without a `select` attribute, the default value `child::node()` is inserted.

We then α-convert all variables and parameters to make code motion transformations easier in the later steps. More precisely, all variables and parameters are renamed consistently such that their uses still refer to the original declarations but they all have unique names. This is straightforward since the declarations in XSLT have lexical scope. Also, all qualified XML names are changed such that XSLT instructions use the default namespace, and the prefixes `in` and `out` identify the input and output language, respectively.

XPath location paths in variable and parameter definitions appearing at top-level are prefixed by `/` to ensure that evaluation remains starting at the document root, even if the definitions are moved away from top-level in later steps. Likewise, all top-level uses of functions that rely on the context node, size, or position are changed appropriately.

We then desugar certain constructs to more basic ones. This includes the following steps:

- For all occurrences of `include`, `import`, and `apply-imports`, the external definitions are inserted into the main stylesheet. For the `import` mechanism, we use `priority` and `mode` to ensure proper overriding. Imported template rules that cause naming conflicts by this transformation are renamed consistently.

- All XPath expressions that use abbreviated syntax are expanded (except for some abbreviated syntax in patterns where expansion is not allowed)—for example, `//` is changed to `/descendant-or-self::node()/`. Also, implicit coercions are made explicit.

- Each use of a variable is replaced by its definition. There are two exceptions, though. First, to preserve evaluation contexts, variables that are used inside a `for-each` instruction but declared outside are instead converted to template parameters, which we treat later. Second, for variables of type result tree fragment, the situation is a little more complicated. If such a variable appears in a `copy-of` instruction as in

  ```
  <copy-of select="$x"/>
  ```

  which is a common use of this instruction, then the entire `copy-of` instruction is replaced by the definition of x. Result tree fragment variables used in other contexts are unchanged for now. Note that this step only involves variables; parameters are treated later.

- All literal result elements and their attributes are converted to `element` and `attribute` instructions, and all text nodes and all occurrences of `text` are converted to `value-of`. Each use of `use-attribute-sets` is replaced by the corresponding `attribute` definitions. Furthermore, each `if` instruction is converted to the more general `choose`, and in each `choose` instruction, if no `otherwise` branch is present, one with an empty template is inserted.

- The instructions `for-each`, `call-template`, `copy-of`, and `copy` can all be reduced to `apply-template` instructions and new template rules. For example, every `for-each` instruction is desugared as follows:

```
<for-each select="exp"> sort templ </for-each>
```

  where *sort* is a sequence of `sort` instructions and *templ* is the template part, is converted to

```
<apply-templates select="exp" mode="x">
  sort
</apply-templates>
```

  where x is a unique mode name, and the template part is moved to a new template rule:

```
<template match="child::node()|attribute::*|/"
          mode="x" priority="0">
  templ
</template>
```

  The value of `priority` is irrelevant here. The soundness of this reduction relies on the assumption that namespace nodes are not selected, as mentioned earlier. If the template *templ* uses any locally declared parameters, then these are forwarded by adding corresponding `with-param` and `param` instructions to the new `apply-templates` instruction and the template rule.

- All `call-template` instructions are handled similar to `for-each` instructions and we omit the details.

- Every `copy-of` instruction is desugared according to the type of its `select` expression. However, if the expression involves parameters, then we generally do not know the type statically, in which case we leave the `copy-of` instruction unmodified for now. Otherwise, if the type is string, boolean, or number, then the instruction is changed to a `value-of` instruction. If the type is node-set, then the `copy-of` instruction instead becomes

```
<apply-templates select="exp" mode="x"/>
```

  where x is a unique mode name, and a new template rule is constructed using a `copy` instruction:

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

- To desugar a `copy` instruction

  ```
  <copy> templ </copy>
  ```

  we convert it to

  ```
  <apply-templates select="self::node()" mode="x">
  ```

  and add some new template rules to accommodate for the different kinds of nodes that may be copied:

  ```
  <template match="/" mode="x" priority="0">
    templ
  </template>

  <template match="child::*" mode="x" priority="0">
    <element name="{name()}"> templ </element>
  </template>

  <template match="attribute::*" mode="x" priority="0">
    <attribute name="{name()}">
      <value-of select="string(self::node())"/>
    </attribute>
  </template>

  <template match="child::text()" mode="x" priority="0">
    <value-of select="string(self::node())"/>
  </template>

  <template match="child::comment()" mode="x" priority="0">
    <comment>
      <value-of select="string(self::node())"/>
    </comment>
  </template>

  <template match="child::processing-instruction()"
            mode="x" priority="0">
    <processing-instruction name="name()">
      <value-of select="string(self::node())"/>
    </processing-instruction>
  </template>
  ```

  Again, x is a fresh mode name, which we use to tie together the new `apply-templates` instruction and the template rules. Any locally declared parameters being used in the original template are forwarded by adding corresponding `with-param` and `param` instructions.

At this stage, we unify template rules that are identical except for different values of `mode`. This is strictly not necessary, but it helps in limiting the size of the simplified stylesheet.

6.2 Approximating Simplifications

The second category introduces approximations. In particular, we do not wish to model computations of strings or booleans in XPath expressions. To model unknown values, we introduce three special extension functions, xslv:unknown-String(), xslv:unknownBoolean(), and xslv:unknownRTF(), which return an arbitrary string, boolean, or result tree fragment, respectively, at each invocation.

Regarding instructions involved with computation of character data or attribute values, we preserve only value-of instructions whose select expression is either a constant string, string(self::node()), or string(attribute::*a*) for some name *a* (the latter two may originate from copy instructions or from explicitly moving attribute values from input to output without modifications). Other expressions are replaced by <value-of select="xslv:unknown String()"/>. Likewise, occurrences of strip-space, preserve-space, and decimal-format are simply removed, and number is treated as value-of. Since DTD has limited control over text values (only simple constraints on attribute values can be expressed), these approximations seem plausible, and our experiments in Section 12 support the choices made here.

Regarding boolean expressions, we replace all test expressions in when constructs by xslv:unknownBoolean(). This is usually sufficient since there is rarely a correlation between the choose branch taken and the name of the parent element in the output. (In fact, in the 187,015 lines of XSLT fra Section 4, this never occurs.)

Also, all location step predicates (that is, the contents of [...] in location steps) are replaced by xslv:unknownBoolean(). We discuss possible improvements of this simplification in Section 7.5.

The later phases of our analysis do not work well with computed (that is, non-constant) element or attribute names. Fortunately, such constructs are uncommon, except for the expression name(), which, for example, arises in the desugaring of copy instructions. To this end, we replace the value of each name attribute occurring in an attribute or element instruction by {xslv:unknownString()}, unless the value is {name()} or a constant string. (In the XSLT stylesheets mentioned in Section 4, this approach handles all but 12 of 940 element names and all but 10 of 5,904 attribute names.)

For sort instructions, we do not model the sorting criteria but merely change their select expressions to xslv:unknownString(). Also, uses of result tree fragment variables that have not been handled earlier are simply replaced by xslv:unknownRTF().

We approximate each use of the key function by replacing it by //*M* where *M* is the match expression of the corresponding key declaration. All key declarations can then be removed. Similarly, each use of the id function is replaced by //child::e_1|....|//child::e_n where the e_i's are the names of elements in the input schema that contain an ID attribute. (In the XSLT samples mentioned in Section 4, the 10,768 select attributes only contain 25 occurrences of the key function and 16 occurrences of the id function.)

All occurrences of `processing-instruction` and `comment` instructions are removed. However, since elements whose content model are `EMPTY` according to the DTD schema are not even allowed to contain processing instructions or comments, we issue a warning in case the stylesheet contains a literal result element that has this content model but is not empty. (This never occurs in our mining samples.)

Finally, we look at parameters. As mentioned, we distinguish between local and global parameters. At this stage, parameters can be used only in `select` expressions in `apply-templates` and `copy-of` instructions and in assignments to other parameters via `param` and `with-param`.

Global parameters pose an obvious problem to the validation task: if such parameters may end up in the result document, then we clearly cannot statically guarantee validity in the way the validation challenge was defined in Section 3.3. Not even the types of the actual parameters are known until runtime. Typically, however, global parameters occur in, for example, `value-of` instructions and thus have been approximated by `xslv:unknownString()`. We ensure that all remaining uses of global parameters will be reported as potential validity errors by reducing them to the instruction `<copy-of select="xslv:unknownRTF()"/>`. Fortunately, in our mining samples this happens a total of zero times, so this does not appear to be a significant problem in practice.

Local parameters are in most cases more manageable. We handle these with a simple flow-insensitive analysis as follows. For each parameter name p, all `param` and `with-param` assignments to p are collected. If p is used in, for example, an `apply-templates` instruction we make a `choose` instruction with a branch for each possible assignment to p, containing a copy of the `apply-templates` instruction, and the parameters are then desugared as if they were variables, as explained earlier. The only remaining problem is cycles of `param` and `with-param` instructions, that is, assignments to a parameter p that directly or indirectly use p itself. In this rather obscure case we consider the possible types of the result and approximate the parameter use correspondingly using either `xslv:unknownString()`, `xslv:unknownBoolean()`, or `xslv:unknownRTF()`.

At this point, the stylesheet has been simplified to a core language that we can focus the analysis on. Note that each approximation step is conservative in the sense that if the resulting stylesheet is valid then so is the original one (but not necessarily the opposite).

6.3 Reduced XSLT

The resulting simplified stylesheet uses only a small subset of the XSLT constructs:

- `template` rules that always use `match` and `priority`, and potentially also `mode` instructions;
- `apply-templates` with `select`, and potentially also with `mode` and `sort` instructions;

```
stylesheet ::= <stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
                           xmlns:xslv="http://www.brics.dk/XSLTValidator"
                           xmlns:in="input-namespace"
                           xmlns:out="output-namespace"
                           version="1.0">
               templaterule
               </stylesheet>

templaterule ::= <template match="pattern" priority="number" ( mode="qname" )? >
                 template
                 </template>

template ::= ( templateinstruction )*

templateinstruction ::= applytemplates | element | attribute |
                        valueof | choose | anything

applytemplates ::= <apply-templates select="exp" ( mode="qname" )? >
                   ( sort )?
                   </apply-templates>

element ::= <element name="name" ( namespace="ns" )? > template </element>

attribute ::= <attribute name="name" ( namespace="ns" )? > valueof </attribute>

valueof ::= <value-of select="stringexp"/>

choose ::= <choose> ( when )* <otherwise> template </otherwise> </choose>

when ::= <when test="xslv:unknownBoolean()"> template </when>

unknown ::= <copy-of select="xslv:unknownRTF()"/>

sort ::= <sort select="xslv:unknownString()"/>

name ::= string | {name()} | {xslv:unknownString()}

stringexp ::= 'string' | xslv:unknownString() |
              string(self::node()) | string(attribute::qname)
```

In this grammar, *pattern* is a reduced template pattern, *exp* is a reduced XPath expression, *qname* is a *QName*, *number* is a number, *ns* is a namespace, and *string* is any string. We use the notation “?” and “*” representing “optional” and “zero-or-more occurrences”, respectively.

Figure 4: Syntax of Reduced XSLT.

- `choose` where each branch condition is `xslv:unknownBoolean()`;
- `sort` criteria are always `xslv:unknownString()`;
- `attribute` and `element` whose `name` is either a constant, `{name()}`, or `{xslv:unknownString()}`, and where the contents of `attribute` is a single `value-of`;
- `value-of` where the `select` expression is either a constant string, `xslv:unknownString()`, `string(self::node())`, or `string(attribute::`*a*`)` for some name *a*; and
- `copy-of` where the `select` expression is `xslv:unknownRTF()`.

Furthermore, use of syntactic sugar and coercions in XPath expressions is eliminated, all location step predicates are changed to `xslv:unknownBoolean()`, and there are no variables or parameters left. The syntax of the resulting language, Reduced XSLT, is provided in Figure 4.

As mentioned, we use a few special extension functions: `xslv:unknownString()`, `xslv:unknownBoolean()`, and `xslv:unknownRTF()` to represent information that has been abstracted away.

Although tedious, the entire simplification phase is straightforward to implement, compared to implementing a full XSLT processor. Obviously, this phase makes the subsequent analysis simpler, and, as argued above and substantiated further in Section 12, it causes no significant loss of precision of the validity analysis on typical stylesheets.

Example

Continuing the example from Section 1, we show what the simplified version of the stylesheet looks like:

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

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

```
  </element>
</template>

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

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

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

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

In the above, we have numbered each template rule and each `apply-templates` and `value-of` instruction. These numbers will be used when we continue the running example in the later phases of the static validation.

7 Flow Analysis

To be able to produce a precise summary graph that represents the possible output of the transformation, we begin by analyzing the flow of the stylesheet.

Given a reduced XSLT stylesheet S and a DTD schema D_{in}, we wish to determine for each `apply-templates` instruction in S the possible target template rules. That is, assuming that $X \in \mathcal{L}(D_{in})$, which templates may be instantiated when processing this particular `apply-templates` instruction on input document X using S?

A *flow edge* is an edge from an `apply-templates` instruction to a possible target template rule. In addition to finding flow edges, we also determine where in S processing may start, that is, which templates may be instantiated when the document root node is processed. We call these the *entry templates*.

For each template rule, we furthermore need to know the types and names of the possible context nodes when the template is instantiated during processing of S on some input document $X \in \mathcal{L}(D_{in})$. Assume that D_{in} describes element names $\mathcal{E}$ and attribute names $\mathcal{A}$. Define

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

representing the types and names of the possible context nodes. A value from $\mathcal{E}$ represents an element of that name; similarly, the values in $\mathcal{A}$ represent attribute names; the dummy names root, pcdata, comment, and pi represent the document root node, arbitrary character data nodes, comment nodes, and processing instructions, respectively. Note that attributes are modeled as pairs of attribute names and element names, which allows distinction between attributes that have the same name but belong to elements with different names, reflecting the way constraints are associated with attributes in DTD. The *context set* of a template rule n is a subset of Σ representing the possible context nodes. For later use, we also define the subset $\Gamma = \mathcal{E} \cup \{\mathsf{pcdata}, \mathsf{comment}, \mathsf{pi}\}$ corresponding to the types that can appear in element contents.

Finally, each flow edge is labeled with a map that for each possible context node type returns a subset of Σ that represents the nodes that may be selected from the associated expression.

Given that the stylesheet S contains a set of template rules T_S and a set of `apply-templates` instructions A_S, we can formally define a flow graph G as follows:

$$G = (C, F)$$

where

$C : T_S \to 2^\Sigma$ describes the *context sets* for the template rules, and
$F : A_S \times T_S \to (\Sigma \to 2^\Sigma)$ describes the *edge flow*.

A pair $(a, t) \in A_S \times T_S$ where $F(a, t)(\sigma) = \emptyset$ for all $\sigma \in \Sigma$ corresponds to not having an edge from a to t. An entry template t is one where $\mathsf{root} \in C(t)$.

In the following, we describe how this information is obtained statically. We settle for a conservative approximation meaning that the sets we produce may be too large but never too small compared to the possible runtime behavior.

Example

The desired flow graph for our example stylesheet should, in particular, show that $\mathsf{root} \in C(\mathtt{5})$ and $\mathtt{group} \notin F(\mathtt{1.2}, \mathtt{2})(\mathtt{registrations})$, in other words: template rule `5` is an entry template, and `group` elements never flow from instruction `1.2` to template rule `2` starting from a `registration` element. We continue the example in Section 7.5 and show that the flow graph being constructed does indeed have these properties.

7.1 The Fixed Point Algorithm

Our flow analysis is based on a fixed point algorithm, which computes the least solution to a system of constraints. First, we find the entry templates:

(1) $\mathsf{root} \in C(t)$ if the `match` expression of t matches the root node.

This property can be checked for a given `match` expression in the same way a normal XSLT processor finds out where to start.

Second, flow is propagated. In the following sections we introduce a concrete approach that provides a function Φ that conservatively approximates the possible flow. This function is specified as follows. We use $select_a$ to denote the `select` expression of an `apply-templates` instruction a, and $match_t$ denotes the `match` expression of a template rule t. Assume that the `apply-templates` instruction a is being evaluated during processing of S on an input document in D_{in} with a current context node of type σ, and control as a result is transferred to the template rule t' (see Figure 5). The function $\Phi(\sigma, select_a, match_{t'}, match_t) \subseteq \Sigma$ then returns an upper approximation of the set of possible types of the new context nodes. This gives rise to the following constraint:

(2) $\sigma \in C(t) \Rightarrow \Phi(\sigma, select_a, match_{t'}, match_t) \subseteq F(a, t')(\sigma)$ where the template rule t contains the `apply-templates` instruction a.

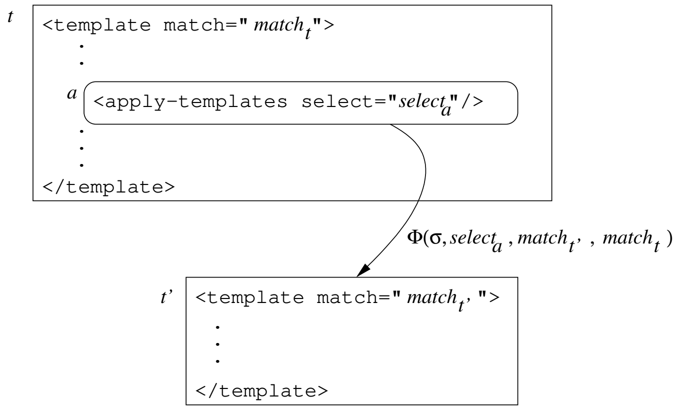


Figure 5: Computing flow from an `apply-templates` instruction.

Note that we allow Φ to depend on $match_t$. We discuss later the influence this has on the analysis precision.

Finally, flow through edges is accumulated in the context sets of the targets:

(3) $F(a,t)(\sigma) \subseteq C(t)$.

Clearly, a simple iterative process will produce the desired solution; the only challenge is to compute Φ with sufficient precision.

7.2 Abstract Evaluation of Location Paths on DTD Schemas

To be able to compute the Φ function, we will need an algorithm for abstractly evaluating an XPath location path on a DTD schema. Given a node type $\sigma \in \Sigma$ and an XPath location path p, the algorithm finds (an upper approximation of) the set $\Delta(\sigma, p)$ of all node types $\delta \in \Sigma$ that satisfy the following requirement:

> There exists an XML document $X \in \mathcal{L}(D_{in})$ with nodes x and y such that $x \overset{p}{\rightsquigarrow} y$, x is a node of type σ, and y is a node of type δ.

We proceed as follows. First, from D_{in}, we construct a directed graph, called the *axis graph for D_{in}*, with a node for each symbol in Σ and with edges for each axis in XPath: if $\sigma, \sigma' \in \mathcal{E}$ and σ' occurs in the content model of σ according to D_{in} then the graph has a *child* edge from σ to σ', and similarly for the other node types and the axes `parent`, `attribute`, `following-sibling`, and `preceding-sibling`. Edges for the `descendant` axis are computed as the transitive closure of the *child* edges, and similarly for the `descendant-or-self`, `ancestor`, and `ancestor-or-self` axes. Edges for the `self` axis are made from every node to itself. The `following` and `preceding` axes are handled very abstractly by edges between all nodes in Γ. (These two axes are used in only 0.7% of the `select` expressions in our mining samples.)

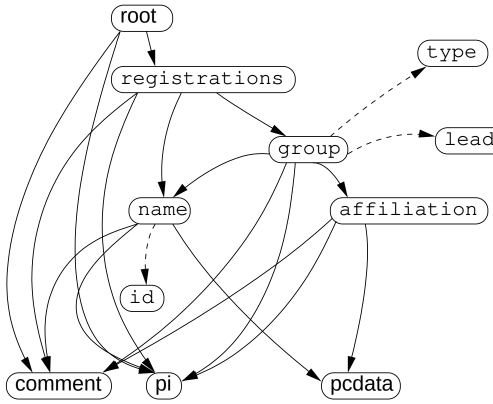


Figure 6: Axis graph for the example DTD schema. (To keep the figure reasonably simple, only the *child* and *attribute* edges are shown; the former are solid and the latter are dashed.)

As an example, the axis graph for the DTD schema from Section 3.1 looks as shown in Figure 6 (provided that we ignore all but the *child* and *attribute* edges).

Now, the approximation of $\Delta(\sigma, p)$ is computed by abstractly evaluating p on the axis graph, starting at the σ node: axis steps correspond to moving along the appropriate edges, and node tests filter the intermediate results. Predicates are simply ignored—this may give a loss of precision but only on the safe side. (The technique could easily be extended to model nested location paths, that is, predicates containing location paths, but the current precision appears to be sufficient.)

As an example, computing $\Delta(\texttt{name}, \texttt{../*})$ for the example DTD schema results in the set $\{\texttt{name}, \texttt{group}, \texttt{affiliation}\}$.

Notice the approximation that is taking place, even if ignoring the `following` and `preceding` axes: for example, the location path `parent::x/child::y/parent::z` may for some schemas yield a nonempty result, but clearly, the exact result will always be empty. Still, for more natural location paths, this abstract evaluation approach gives reasonable precision.

7.3 Select–Match Compatibility

We now look into computing the Φ function. Considering the semantics of `apply-templates`, we can reformulate the specification of Φ from Section 7.1 using a compatibility condition on XPath location paths:

> $\sigma' \in \Phi(\sigma, select_a, match_{t'}, match_t)$ if there exists an XML document $X \in \mathcal{L}(D_{in})$ with nodes x_1, x_2, x_3, x_4 such that $x_1 \overset{match_t}{\leadsto} x_2$, $x_2 \overset{select_a}{\leadsto} x_3$, $x_4 \overset{match_{t'}}{\leadsto} x_3$, x_2 is a node of type σ, and x_3 is a node of type σ'.

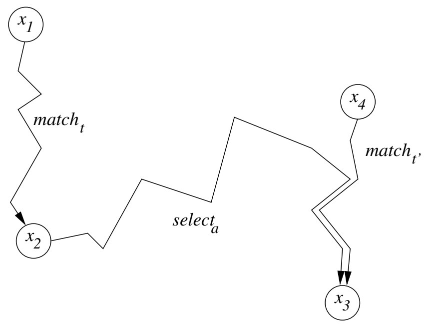


Figure 7: The `select`–`match` compatibility condition.

Intuitively, when the `apply-templates` instruction is processed, x_2 can only be a context node if it matches $match_t$, and when $select_a$ is evaluated starting from x_2 then some node x_3 in the resulting node set must match $match_{t'}$ in order for x_3 to be a possible target (see Figure 7). Note that this is a necessary but not always sufficient condition for having control-flow from a to m.

Notice an analogy with k-CFA analysis [19], in particular 1-CFA: our analysis of an `apply-templates` instruction may depend on information from the caller in the form of the $match_t$ expression. 0-CFA would then correspond to not considering $match_t$, 2-CFA corresponds to also considering the `match` expression of template rules that have a flow edge to t, and so on. Hence, there is an opportunity for tuning the precision, but our experiments indicate that the present choice of context sensitivity is adequate.

We can simplify the condition above by combining the expressions and node tests as follows. Unfortunately, XPath does not make it easy to test that the current node is an element or attribute with a certain name or that it is a root node, so we first introduce some syntactic sugar:

```
element(σ) = self::*[name()='σ']
attribute(a) = self::node[name()='a' and
                          not(self::*) and
                          not(self::processing-instruction())]
root() = self::node()[not(parent::node())]
```

(One may ask why we choose to go into so much trouble to stay within the XSLT language rather than simply introduce a clean intermediate language; the answer is that (1) in this way, we do not have to explain the semantics of a new language, and (2) it is more likely that the techniques we develop will be of a more general use when we stay close to the existing language.) Now, define

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

$$\beta = match_{t'}$$

where the *type* function encodes the node type:

$$type(\sigma) = \begin{cases} \texttt{self::element(}\sigma\texttt{)} & \text{if } \sigma \in \mathcal{E} \\ \texttt{self::attribute(}a\texttt{)} & \\ \quad \texttt{[parent::*[name()='}e\texttt{']]} & \text{if } \sigma = (a,e) \in \mathcal{A} \times \mathcal{E} \\ \texttt{self::root()} & \text{if } \sigma = \textsf{root} \\ \texttt{self::text()} & \text{if } \sigma = \textsf{pcdata} \\ \texttt{self::comment()} & \text{if } \sigma = \textsf{comment} \\ \texttt{self::processing-instruction()} & \text{if } \sigma = \textsf{pi} \end{cases}$$

The compatibility condition is then seen to be equivalent with the following simpler requirement:

> $\sigma' \in \Phi(\sigma, select_a, match_{t'}, match_t)$ if there exists an XML document $X \in \mathcal{L}(D_{in})$ with nodes x_1, x_3, x_4 such that $x_1 \overset{\alpha}{\rightsquigarrow} x_3$, $x_4 \overset{\beta}{\rightsquigarrow} x_3$, and x_3 is a node of type σ'.

Compared to the earlier condition, we have here combined the $match_t$ and $select_a$ expressions and the type requirement on the x_2 node into a single expression. The result shows that obtaining the flow information essentially amounts to checking that two XPath location paths, α, β, are "compatible" relative to a schema D_{in}.

As an aside, we can also use this as an alternative technique to easily find the entry templates: simply check for each template rule whether its `match` expression is compatible with the `select` expression `/` (for some arbitrary value of σ).

Every DTD schema defines a regular tree language and hence can be captured as a formula in monadic second-order logic on trees (M2L-Tree) [24]. XPath location paths, as they appear at this stage, can also be encoded into M2L-Tree, essentially in the same way auxiliary pointers are encoded in graph types [26]. This sketch of an argument indicates that the problem of checking compatibility is decidable; however, based on our previous experience with M2L-Tree [25], we foresee that an algorithm based entirely on this approach will not be sufficiently efficient in practice.

Instead, we suggest, based on the statistical results from Section 4, a more pragmatic approach that does not involve regular tree languages, as described in the following.

7.4 Computing Flow

As pointed out in Section 4, more that 90% of all `select` expressions use only the downward axes. All `match` expressions are in XSLT always constrained to these axes also. For the remaining `select` expressions, which use a non-downward axis (`parent`, `ancestor`, `ancestor-or-self`, `following`, `preceding`, `following-sibling`, or `preceding-sibling`), we approximate the expression as follows. Assume that the expression has the form $s_1/s_2/\ldots/s_n$ and that s_i is the rightmost location step containing a non-downward axis. We then compute $\{\sigma_1, \ldots, \sigma_m\} = \Delta(\sigma, s_1/s_2/\ldots/s_i)$ and rewrite the expression to

$$//\mathit{type}(\sigma_1)/s_{i+1}/\ldots/s_n \;\mid\; \ldots \;\mid\; //\mathit{type}(\sigma_m)/s_{i+1}/\ldots/s_n$$

which is a union of downward expressions. Of course, if $\{\sigma_1, \ldots, \sigma_m\}$ contains all of, for instance, $\mathcal{E}$, then the resulting expression can be simplified accordingly. The approximation of the non-downward steps intuitively loses track of concrete nodes but retains their types.

As an example, the non-downward expression `../a/b` will, under the assumption that $\Delta(\sigma, \texttt{..}) = \{\texttt{c}, \texttt{d}\}$ where $\texttt{c}, \texttt{d} \in \mathcal{E}$, be approximated by the following expression:

```
//element(c)/a/b | //element(d)/a/b
```

At this stage, we can assume that α and β use only the downward axes, which we exploit in the following.

Define a *valid downward path relative to D_{in}* as a finite string w over the alphabet Σ starting with root and satisfying the property that for every symbol σ in w, the successor σ', if present, respects the DTD schema D_{in}:

- root only appears as the very first symbol in the string;
- if $\sigma = \mathsf{root}$ then $\sigma' \in \{root(D_{in}), \mathsf{comment}, \mathsf{pi}\}$;
- if both $\sigma \in \mathcal{E}$ and $\sigma' \in \mathcal{E}$ then σ' may appear as child of σ according to the content model of σ;
- if $\sigma \in \mathcal{E}$ and $\sigma' = \mathsf{pcdata}$, then the content model of σ permits character data;
- if $\sigma \in \mathcal{E}$ and $\sigma' = (\sigma'_A, \sigma) \in \mathcal{A} \times \mathcal{E}$, then σ elements may have attributes named σ'_A according to the schema; also, if $\sigma' = (\sigma'_A, \sigma'_E)$ then $\sigma = \sigma'_E$;
- if $\sigma \in \mathcal{E}$ and the content model of σ is `EMPTY`, then σ has no successors (not even a comment); and
- pcdata, comment, and pi symbols and all symbols from $\mathcal{A} \times \mathcal{E}$ have no successors;

The set of such paths forms a simple regular string language $\Pi(D_{in})$. A DFA (deterministic finite-state automaton) representing this language can easily be constructed in linear time in the size of D_{in}.

As an example,

$$\mathsf{root}\ \texttt{registrations}\ \mathsf{group}\ (\texttt{type}, \mathsf{group})$$

is a valid downward path relative to the example DTD schema shown in Section 3.1.

The downward XPath expressions α and β can similarly be encoded as regular expressions over Σ, as defined by the function $R(p)$ below. We aim for a regular expression that has the property that if $x \overset{p}{\rightsquigarrow} y$ for some nodes x, y then the path from x to y corresponds to a string over Σ that matches $R(p)$, and vice versa.

$$R(p) = \begin{cases} R(p_1) + R(p_2) & \text{if } p \text{ has the form } p_1 \texttt{ | } p_2 \\ R(q)R^{axis}(a) \cap \Sigma^* R_a^{test}(t) & \text{if } p \text{ has the form } q\ a\texttt{::}t \\ & \quad \text{where } a\texttt{::}t \text{ is the rightmost location step} \\ \mathsf{root} & \text{if } p \text{ is } \texttt{/} \\ \epsilon & \text{if } p \text{ is empty} \\ R(q) & \text{if } p \text{ has the form } q\texttt{/} \end{cases}$$

$$R^{axis}(a) = \begin{cases} \epsilon & \text{if } a = \texttt{self} \\ \Gamma & \text{if } a = \texttt{child} \\ \mathcal{E}^*\Gamma & \text{if } a = \texttt{descendant} \\ \mathcal{E}^*\Gamma + \epsilon & \text{if } a = \texttt{descendant-or-self} \\ \mathcal{A} \times \mathcal{E} & \text{if } a = \texttt{attribute} \end{cases}$$

$$R_a^{test}(t) = \begin{cases} t & \text{if } t \in \mathcal{E} \\ \{t\} \times \mathcal{E} & \text{if } t \in \mathcal{A} \\ \mathcal{E} & \text{if } t = \texttt{*} \text{ and } a \neq \texttt{attribute} \\ \mathcal{A} \times \mathcal{E} & \text{if } t = \texttt{*} \text{ and } a = \texttt{attribute} \\ \Sigma & \text{if } t = \texttt{node()} \\ e & \text{if } t = \texttt{element(}e\texttt{)} \\ (b, e) & \text{if } t = \texttt{attribute(}b\texttt{)[parent::*[name()='}e\texttt{']]} \\ \mathsf{root} & \text{if } t = \texttt{root()} \\ \mathsf{pcdata} & \text{if } t = \texttt{text()} \\ \mathsf{comment} & \text{if } t = \texttt{comment()} \\ \mathsf{pi} & \text{if } t = \texttt{processing-instruction()} \\ & \quad \text{or } t = \texttt{processing-instruction(}x\texttt{)} \end{cases}$$

The function R^{axis} models axes, and R_a^{test} models a node test relative to an axis. We here assume the non-abbreviated form of XPath expressions (for example, `a/b` is an abbreviation of `child::a/child::b`). We can safely ignore all predicates here since we are computing an upper approximation. We do, however, consider the special XPath predicates used above in the `attribute(`a`)` construct and in the definitions of the syntactic sugar (`element`, `attribute`, and `root`) in order to increase precision.

The construction of the regular expressions is complicated by the `self` and `descendant-or-self` axes, which permit multiple location steps to examine the same node. Also, the use of intersections is not typical in regular expressions, but, of course, we stay within the regular languages.

A few examples (the rightmost simplified expressions are included to help readability):

$$R(\texttt{/a}) = \textsf{root}\ \Gamma \cap \Sigma^* \ \texttt{a} = \textsf{root}\ \texttt{a}$$

$$R(\texttt{a/b}) = (\epsilon\ \Gamma \cap \Sigma^*\ \texttt{a})\ \Gamma \cap \Sigma^*\ \texttt{b} = \texttt{a b}$$

$$\begin{aligned} &R(\texttt{//element(c)/a/b | //element(d)/a/b}) \\ &\quad = (((\textsf{root}\ \Gamma^* \cap \Sigma^*\ \Sigma)\ \Gamma \cap \Sigma^*\ \texttt{c})\ \Gamma \cap \Sigma^*\ \texttt{a})\ \Gamma \cap \Sigma^*\ \texttt{b} + \\ &\qquad (((\textsf{root}\ \Gamma^* \cap \Sigma^*\ \Sigma)\ \Gamma \cap \Sigma^*\ \texttt{d})\ \Gamma \cap \Sigma^*\ \texttt{a})\ \Gamma \cap \Sigma^*\ \texttt{b} \\ &\quad = \textsf{root}\ \Sigma^*(\texttt{c} + \texttt{d})\ \texttt{a b} \end{aligned}$$

We can now compute an approximation of the Φ function specified on page 27 as follows by combining the regular languages $R(\alpha)$, $R(\beta)$, and $\Pi(D_{in})$:

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

iff

$$w\sigma' \in \Sigma^* R(\alpha) \cap \Sigma^* R(\beta) \cap \Pi(D_{in}) \text{ for some } w \in \Sigma^*$$

The intersection of $\Sigma^* R(\alpha)$ and $\Sigma^* R(\beta)$ corresponds to downward paths that are feasible with both α and β. The intersection with $\Pi(D_{in})$ ensures that we only consider valid input documents. The last symbol in each of the resulting strings represents the type of the destination context node for the `apply-templates` instruction.

This can all be computed with well-known algorithms for regular expressions and finite-state automata [17]. Efficiency can be improved by exploiting the special structure of the regular expressions being used.

In summary, this approach of computing the flow involves conservative approximations only in the modeling of non-downward axes and XPath predicates.

7.5 Refinements

The following refinements of the above techniques allow us to improve precision or performance of the flow analysis.

Context and Schema Insensitive Analysis

In the algorithm described above, a `select`–`match` compatibility check is performed for every `match` expression each time a context set is extended. We can improve efficiency by first performing a less precise analysis that filters out a number of infeasible flow edges. First, we define a variant of α that does not consider the evaluation context:

$$\alpha' = \begin{cases} \mathit{select}_a & \text{if } \mathit{select}_a \text{ starts with } \texttt{/} \\ \mathit{match}_t\texttt{/}\mathit{select}_a & \text{otherwise} \end{cases}$$

If the expression α' uses non-downward axes, we approximate it much like before but without considering the evaluation context: assuming that α' has the form $s_1\texttt{/}s_2\texttt{/}\ldots\texttt{/}s_n$ and that s_i is the rightmost location step containing a non-downward axis, we change the expression to $\texttt{//}s_{i+1}\texttt{/}\ldots\texttt{/}s_n$. Now, if

$$\Sigma^* R(\alpha') \cap \Sigma^* R(\mathit{match}_{t'}) = \emptyset$$

then we can safely set $F(a, t')(\sigma) = \emptyset$. Clearly, this can be checked more quickly than the full version, which also depends on δ and D_{in}.

Our experiments indicate that this extension in a few cases causes a small slowdown but in many cases results in dramatic performance improvements.

Handling Modes

The `mode` attributes are modeled by two small modifications of the procedure described in Section 7.4. First, a template rule can only be marked as an entry if its `mode` is absent. Second, a flow edge is never added if the `mode` of the `apply-templates` instruction is different from that of the target template rule—that is, the edge flow function F always yields $\emptyset$ in this case.

Handling Priorities

By considering the `priority` attributes of the template rules, we can improve precision of the flow analysis by omitting flow from a to to a template rule t_1 if we can guarantee that another template rule t_2 with higher priority will always be applicable whenever t_1 is. As a static approximation, we consider the following problem.

> Let $\mathit{priority}(t)$ denote the value of the `priority` attribute of template rule t. If, for two flow edges $(a, t_1), (a, t_2) \in A_S \times T_S$ and context node types $\sigma, \sigma' \in \Sigma$, both of the following conditions are satisfied, then σ' nodes can never flow along (a, t_2) when a is processed with a context node of type σ:
>
> - $\mathit{priority}(t_1) > \mathit{priority}(t_2)$; and
> - for every XML document $X \in \mathcal{L}(D_{in})$ with nodes x_1, x_2, x_3, if $x_1 \overset{\mathit{match}_{t_2}}{\rightsquigarrow} x_2$, x_2 has type σ', x_3 has type σ, and $x_3 \overset{\mathit{select}_a}{\rightsquigarrow} x_2$, then $x_4 \overset{\mathit{match}_{t_1}}{\rightsquigarrow} x_2$ for some x_4.
>
> In other words, we may in this case safely omit the flow σ' in $F(a, t_2)(\sigma)$.

The first condition is, of course, trivial to check. To check the second condition, we apply the techniques developed in Section 7.4. Concretely, if

$$\Sigma^*(R(\mathit{match}_{t_2} \cap \sigma\Sigma^*)) \cap \Sigma^*\sigma' \cap \Sigma^* R(\mathit{select}_a) \cap \Pi(D_{in}) \ \subseteq\ \Sigma^* R(\mathit{match}_{t_1})$$

and $match_{t_1}$ contains no predicates, then the second condition above is satisfied, and this can again be checked with standard automata operations. (Consequently, we do not attempt to exploit `priority` attributes if $match_{t_1}$ does contain predicates, but, on the other hand, predicates in $match_{t_2}$ can simply be ignored here.) If $select_a$ involves non-downward axes, we first rewrite it using the technique described in Section 7.4.

The soundness of this check relies on a property of the definition of the encoding $R(match_{t_1})$: for expressions without predicates, the encoding is *exact* in the sense that $x \overset{p}{\rightsquigarrow} y$ for some nodes x, y iff the path from x to y corresponds to a string over Σ that matches $R(p)$.

As an example, consider a case where $priority(t_1) = 2$, $priority(t_2) = 0.5$, $match_{t_1} = \texttt{c/d}$, $match_{t_2} = \texttt{b/c[@a='42']/*}$, $\sigma = \texttt{e}$, and $\sigma' = \texttt{d}$. With the technique presented above, we can determine—in this case even without considering $\Pi(D_{in})$—that the conditions are satisfied, so $\texttt{d} \notin F(a, t_2)(\texttt{e})$.

Handling Predicates

Currently, we do not exploit relationships between location step predicates in the flow analysis. (Recall from Section 6.2 that we approximate them by `xslv:unknownBoolean()`—however we do introduce a few artificial predicates in the flow analysis that are taken into account.) Obviously, there is room for improvement here, although our experiments indicate that the current level of precision is usually sufficient.

One concrete improvement would be to use a simple theorem prover in the `select`–`match` compatibility checker (see Section 7.3). For example, consider an `apply-templates` instruction with `select` expression `item[@class = 2]`, a template rule with `match` expression `item[not(@class) and @type]` and another with `match` expression `item[@class > 7]`. A theorem prover that can reason about propositional logic and simple arithmetic would be able to determine that neither `match` expression matches the `select` expression, and hence that no flow can occur along the corresponding edges.

Example

For our example stylesheet (see the reduced version in Section 6.3), the flow analysis proceeds as follows. First, the only `match` expression that matches the root node is that of template rule `5`, so initially, the context set map maps `5` to $\{\texttt{root}\}$ and all others to $\emptyset$. We write this as $[\texttt{5} \mapsto \{\texttt{root}\}]$. The edge flow map is initially constantly $\emptyset$, which we write as $[]$.

Template rule `5` contains a single `apply-templates` instruction, `5.1`, so we apply the flow propagation constraint. This requires us to compute $\Phi(\texttt{root}, \texttt{child::node()}, match_{t'}, \texttt{/})$ for each potential destination template rule t'. For $t' = \texttt{1}$, this results in the set $\{\texttt{registrations}\}$; for all others, the result is $\emptyset$. In other words, control may flow from instruction `5.1` to template rule `1`, and the new context node, which is added to the context set of template rule `1`, is of type `registrations` (for such a simple case, this should not be surprising).

$$
\begin{gathered}
([5 \mapsto \{\mathsf{root}\}], [\,]) \\
\downarrow \\
([1 \mapsto \{\texttt{registrations}\}, 5 \mapsto \{\mathsf{root}\}], [(5.1, 1) \mapsto \{\texttt{registrations}\}]) \\
\downarrow \\
([1 \mapsto \{\texttt{registrations}\}, 2 \mapsto \{\texttt{name}\}, 3 \mapsto \{\texttt{group}\}, 5 \mapsto \{\mathsf{root}\}], \\
[(1.2, 2) \mapsto \{\texttt{name}\}, (1.2, 3) \mapsto \{\texttt{group}\}, (5.1, 1) \mapsto \{\texttt{registrations}\}]) \\
\downarrow \\
([1 \mapsto \{\texttt{registrations}\}, 2 \mapsto \{\texttt{name}\}, 3 \mapsto \{\texttt{group}\}, 4 \mapsto \{\texttt{name}\}, 5 \mapsto \{\mathsf{root}\}], \\
[(1.2, 2) \mapsto \{\texttt{name}\}, (1.2, 3) \mapsto \{\texttt{group}\}, (3.4, 4) \mapsto \{\texttt{name}\}, (5.1, 1) \mapsto \{\texttt{registrations}\}])
\end{gathered}
$$

Figure 8: Fixed point computation for the example.

Figure 8 shows the continued fixed point iterations. The last line shows the complete flow, which is also illustrated as a graph in Figure 9. Note that the analysis discovers that the only possible context nodes of template rule `2`, which has the `match` expression `*`, are `name` elements. (Without the modeling of priorities, the flow graph would contain additional edges from `3.4` and `5.1` to `2`.)

8 Summary Graphs

With the information provided by the flow analysis, we can now begin constructing a summary graph that represents the possible output of the stylesheet. The following definition of what summary graphs are is a variation of the one presented in [23]. Due to the highly flexible nature of XSLT (for example, attribute values, attributes, and elements can be constructed separately by XSLT instructions located in distinct template rules), we need a number of modifications of the earlier definition; we describe these after presenting the definition we use here.

A summary graph is defined relative to an input schema D_{in} and an XSLT stylesheet S (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 shall represent the possible elements, attributes, and character data or attribute values that may occur when running the stylesheet. 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 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 $N = N_{\mathcal{E}} \cup N_{\mathcal{A}} \cup N_{\mathcal{T}} \cup N_{\mathcal{S}} \cup N_{\mathcal{C}}$;

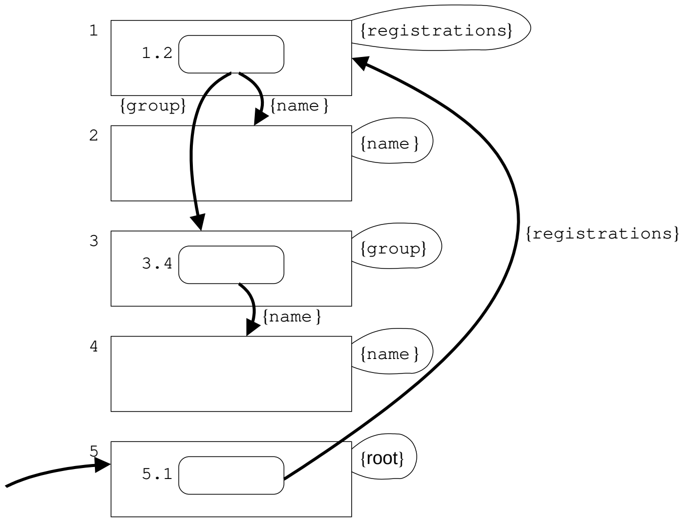


Figure 9: Resulting flow for the example. (Only edges with non-empty flow are shown; the blobs indicate the context sets.)

- $R \subseteq N$ is a set of designated *root nodes*;
- $S : N_{\mathcal{E}} \cup N_{\mathcal{A}} \cup N_{\mathcal{T}} \to 2^{STRINGS}$, where $STRINGS$ is the set of all Unicode strings, defines *node labels*;
- *contains* : $N_{\mathcal{E}} \cup N_{\mathcal{A}} \to N$ defines *contains edges*;
- *seq* : $N_{\mathcal{S}} \to N^*$ defines *sequence edges*; and
- *choice* : $N_{\mathcal{C}} \to 2^N$ 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:

$$\frac{n \in N_{\mathcal{E}} \quad e \in S(n) \quad contains(n) = m \quad m \overset{\mathsf{attr}}{\Rightarrow} \{a_1, \ldots, a_k\} \quad m \overset{\mathsf{cont}}{\Rightarrow} c}{n \Rightarrow \texttt{<}e\ a_1 \ldots a_k\texttt{>}\ c\ \texttt{</}e\texttt{>}}$$

$$\frac{n \in N_{\mathcal{A}} \quad a \in S(n) \quad contains(n) = m \quad m \overset{\mathsf{text}}{\Rightarrow} s}{n \Rightarrow a\texttt{="}s\texttt{"}}$$

$$\frac{n \in N_{\mathcal{T}} \quad s \in S(n)}{n \Rightarrow s}$$

$$\frac{n \in N_{\mathcal{S}} \quad seq(n) = a_1 \dots a_k \quad a_i \Rightarrow b_i \text{ for all } i = 1, \dots, k}{n \Rightarrow b_1 \dots b_k}$$

$$\frac{n \in N_{\mathcal{C}} \quad a \in choice(n) \quad a \Rightarrow b}{n \Rightarrow b}$$

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:

$$\frac{n \in N_{\mathcal{A}} \quad n \Rightarrow a}{n \overset{\mathsf{attr}}{\Rightarrow} \{a\}} \qquad \frac{n \in N_{\mathcal{E}} \cup N_{\mathcal{T}}}{n \overset{\mathsf{attr}}{\Rightarrow} \emptyset}$$

$$\frac{n \in N_{\mathcal{S}} \quad seq(n) = a_1 \dots a_k \quad a_i \overset{\mathsf{attr}}{\Rightarrow} A_i \text{ for all } i = 1, \dots, k}{n \overset{\mathsf{attr}}{\Rightarrow} \bigcup_{i=1}^{k} A_i}$$

$$\frac{n \in N_{\mathcal{C}} \quad a \in choice(n) \quad a \overset{\mathsf{attr}}{\Rightarrow} A}{n \overset{\mathsf{attr}}{\Rightarrow} A}$$

$$\frac{n \in N_{\mathcal{E}} \cup N_{\mathcal{T}} \quad n \Rightarrow x}{n \overset{\mathsf{cont}}{\Rightarrow} x} \qquad \frac{n \in N_{\mathcal{A}}}{n \overset{\mathsf{cont}}{\Rightarrow} \epsilon}$$

$$\frac{n \in N_{\mathcal{S}} \quad seq(n) = a_1 \dots a_k \quad a_i \overset{\mathsf{cont}}{\Rightarrow} b_i \text{ for all } i = 1, \dots, k}{n \overset{\mathsf{cont}}{\Rightarrow} b_1 \dots b_k}$$

$$\frac{n \in N_{\mathcal{C}} \quad a \in choice(n) \quad a \overset{\mathsf{cont}}{\Rightarrow} b}{n \overset{\mathsf{cont}}{\Rightarrow} b}$$

$$\frac{n \in N_{\mathcal{T}} \quad n \Rightarrow x}{n \overset{\mathsf{text}}{\Rightarrow} x} \qquad \frac{n \in N_{\mathcal{E}} \cup N_{\mathcal{A}}}{n \overset{\mathsf{text}}{\Rightarrow} \epsilon}$$

$$\frac{n \in N_{\mathcal{S}} \quad seq(n) = a_1 \dots a_k \quad a_i \overset{\mathsf{text}}{\Rightarrow} b_i \text{ for all } i = 1, \dots, k}{n \overset{\mathsf{text}}{\Rightarrow} b_1 \dots b_k}$$

$$\frac{n \in N_{\mathcal{C}} \quad a \in choice(n) \quad a \overset{\mathsf{text}}{\Rightarrow} b}{n \overset{\mathsf{text}}{\Rightarrow} b}$$

As an example, we can construct a summary graph whose language is the set of `ul` lists with zero or more `li` items that each contain a string from some language L: $N_{\mathcal{E}} = \{1, 5\}$, $N_{\mathcal{A}} = \emptyset$, $N_{\mathcal{T}} = \{6\}$, $N_{\mathcal{S}} = \{3, 4\}$, $N_{\mathcal{C}} = \{2\}$, $R = \{1\}$, $S(1) = \mathtt{ul}$, $S(5) = \mathtt{li}$, $S(6) = L$, $contains(1) = 2$, $contains(5) = 6$, $seq(3) = \epsilon$, $seq(4) = 5\ 2$, and $choice(2) = \{3, 4\}$. This may be shown graphically as in Figure 10.

The present definition of summary graphs differs from the one in [23] in the following ways: first, since we are here not constructing summary graphs in a fixed point process, we do not need labeled gaps or the gap presence map; second, we here have a looser connection between elements, attributes, and attribute values; and third, we can here also represent elements and attributes that do not

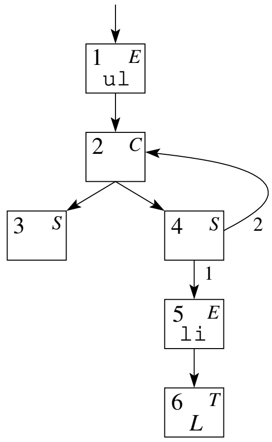


Figure 10: Summary graph representing `ul` lists with zero or more `li` items. The contents of the boxes indicate node ID, node label (for those nodes that have a label), and node type (where E, A, T, S, and C represent element, attribute, text, sequence, and choice node, respectively). For brevity, singleton node labels are written without brackets, and the edge types (contains, sequence, or choice) can be inferred from the context. The edge numbers at the sequence node with ID 4 indicate their order.

have fixed names. Although the basic ideas are not new, these differences permit a more smooth construction of summary graphs, and the existing algorithm for comparing a summary graph and a schema can be modified accordingly, which we return to in Section 11.

9 Construction of Summary Graphs

For each pair of a template rule $t \in T_S$ and a context node type $\sigma \in C(t)$, we construct a summary graph fragment by recursively traversing the structure of the template (see Figure 4):

- A sequence of XSLT instructions in the template becomes a sequence node with an edge to each of the summary graph fragments being constructed for the instructions in the same order.
- An `element` instruction becomes an element node n. Its name $S(n)$ is determined by the `name` attribute of the instruction:
  - for a constant string s, we let $S(n) = \{s\}$;
  - for `{xslv:unknownString()}`, we let $S(n) = STRINGS$; and

– for `{name()}`, we choose $S(n)$ according to the semantics of the `name()` function:

$$S(n) = \begin{cases} \{\sigma\} & \text{if } \sigma \in \mathcal{E} \\ \{a\} & \text{if } \sigma = (a, e) \in \mathcal{A} \times \mathcal{E} \\ \mathit{STRINGS} & \text{if } \sigma = \mathsf{pi} \\ \{\epsilon\} & \text{otherwise} \end{cases}$$

A summary graph fragment represented by a node m is constructed recursively for the contents of the instruction, and $\mathit{contains}(n)$ is set to m.

- An `attribute` instruction becomes an attribute node. Its name and contents are handled as for `element` instructions.

- A `value-of` instruction becomes a text node n. Its label $S(n)$ is determined by the `select` expression:

  – for a constant string s, we let $S(n) = \{s\}$;

  – for `xslv:unknownString()`, we let $S(n) = \mathit{STRINGS}$;

  – for `string(self::node())`, we choose $S(n)$ according to the type of σ:

$$S(n) = \begin{cases} \mathit{values}(D_{in}, e, a) & \text{if } \sigma = (a, e) \in \mathcal{A} \times \mathcal{E} \\ \mathit{STRINGS} & \text{otherwise} \end{cases}$$

  Here, the function $\mathit{values}(D_{in}, e, a)$ returns the set of strings that are valid values of a attributes in e elements according to D_{in}. (This is always a regular language over the Unicode alphabet.)

  – for `string(attribute::a)` for some name `a`, we again choose $S(n)$ according to the type of σ:

$$S(n) = \begin{cases} \mathit{values}(D_{in}, \sigma, \mathtt{a}) & \text{if } \sigma \in \mathcal{E} \\ \{\epsilon\} & \text{otherwise} \end{cases}$$

- A `choose` instruction becomes a choice node with an edge to each of the summary graph fragments being constructed for the branches.

- A `copy-of` instruction (which, at this point, we only use to represent computation of an unknown result tree fragment, as explained earlier) becomes an element node n, $S(n) = \mathit{STRINGS}$, $\mathit{attr}(n) = \emptyset$, and $\mathit{content}(n) = m$ where m is a sequence node with $\mathit{seq}(m) = \epsilon$. That is, this instruction is modeled as an empty element with an unknown name, which is sufficient to trigger a meaningful validity error message in the validation phase described in Section 11.

- An `apply-templates` instruction results in combining the summary graph fragment for t with fragments corresponding to other templates or context node types; we explain how this is done in Section 10.

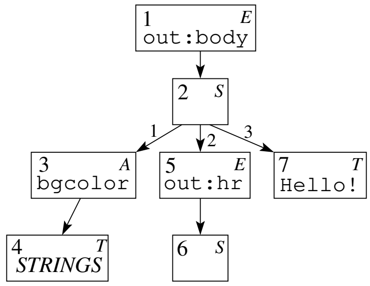


Figure 11: Summary graph fragment for a small template.

If t is an entry template, then the node corresponding to the entire template rule becomes a root node in the summary graph.

As an example, the following small template is translated into the summary graph fragment shown in Figure 11:

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

In general, this translation into summary graphs does not introduce any imprecision that was not already present due to the translation into Reduced XSLT and the approximative nature of the flow analysis.

10 Translating apply-templates Instructions

To connect the summary graph fragments according to the `apply-templates` instructions, we proceed according to Figure 2, which shows how their `select` expressions look in practice. In the following, the numbers in parentheses show how many cases each technique covers.

The Default Expression (31.7%)

The default `select` expression selects all child nodes of the context node. If this is not of type $\mathcal{E}$ (an element) nor of type root, then the possible content is simply modeled by an empty sequence node. Otherwise, we look up the declared content model of σ in D_{in} and build a corresponding summary graph fragment. The content model is a regular expression over symbols from Γ. We first translate this regular language into a similar summary graph construction, using choice nodes, sequence nodes, loop edges, and placeholders for the Γ

symbols. (This technique was introduced in [23].) Each placeholder is then replaced with a choice node that links to the summary graph fragments for the templates that correspond to the possible outgoing flow for that symbol in the `apply-templates` instruction. More precisely, a placeholder $\gamma \in \Gamma$ is replaced with a choice node that links to the summary graph fragment for each template t' where $\gamma \in F(a, t')(\sigma)$.

Projected Contents (+44.8%)

Next, we handle the cases `a`, `*`, `a | b | c`, and `text()`. We here proceed as for the default case, except that the regular expression we consider is obtained from the content model of the context node by deleting the symbols that are not matched by the select expression.

As an example, consider the `apply-templates` with number `3.4` in the expanded version of our running example:

```
<!-- 3 -->
<template match="in:group" priority="0">
  <element name="out:li">
    <element name="out:table">
      ...
      <apply-templates select="in:name"><!-- 3.4 -->
        <with-param name="leader" select="@leader"/>
      </apply-templates>
    </element>
  </element>
</template>
```

Here, the context node has type `in:group`. The regular expression describing the content model is `(in:affiliation,in:name*)`. Restricting to the symbol `in:name` we are left with the projected content `in:name*`. This is represented by the summary graph fragment shown in Figure 12, where the dashed node represents the placeholder for the symbol `in:name`. Consulting the flow graph in Figure 9, we see that there is a single outgoing flow for this symbol to the template rule with number `4` and context node `in:name`. The completed summary graph fragment for the `apply-templates` instruction with number `3.4` is then seen in Figure 13.

Multiple Location Steps (+11.9%)

We now consider the cases `a/b/c` and `/a/b/c`. They are handled one step at a time, by concatenating the summary graph fragments for each step as described above (where the context node type traverses down the sequence). In the case of a leading `/`, the initial context node type is root.

Predicates (+3.4%)

Cases which involve the use of predicates, such as `a[..]`, `a[...]/b[...]/c[...]`, and `/a[...]/b[...]/c[...]`, are handled as the cases without predicates, ex-

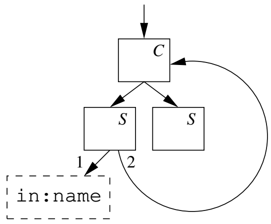


Figure 12: Summary graph fragment modeling projected contents, still with a placeholder node.

cept that the choice nodes representing the possible outgoing flow is extended with an edge to an empty sequence node to model the case where the predicate evaluates to false.

Attributes (+0.6%)

For the case `@a` we have several possibilities. First, if σ is an element and `a` attributes are required (`#REQUIRED` or `#FIXED`) in such elements according to D_{in}, then we know that a single node is selected. We model this with a choice node with outgoing edges to the template rules that correspond to the possible outgoing flow from the `apply-templates` instruction. If the `a` attribute is instead optional (`#IMPLIED` or has a default value), then we do the same but add an edge to an empty sequence node. In other cases, no nodes are selected, which is modeled with an empty sequence node.

Parent and Root (+0.4%)

For the cases `..` and `/`, we know that only a single element is selected (or none, if $\sigma = \mathsf{root}$). Thus, the appropriate summary graph fragment is a choice node with outgoing edges to the template rules that correspond to the possible outgoing flow from the `apply-templates` instruction.

Others (+7.2%)

In all other cases, we resort to constructing a summary graph fragment describing all sequences of possible outgoing flow. For the large subset where the possible names are known (5.6%), this sound approximation can be made more precise by performing a simple cardinality analysis on D_{in}, deciding for each element or attribute how many times it may occur in an output document. We only need to approximate this cardinality with the possibilities `?` (zero or one times), `1` (one time), `*` (zero or more times), or `+` (one or more times).

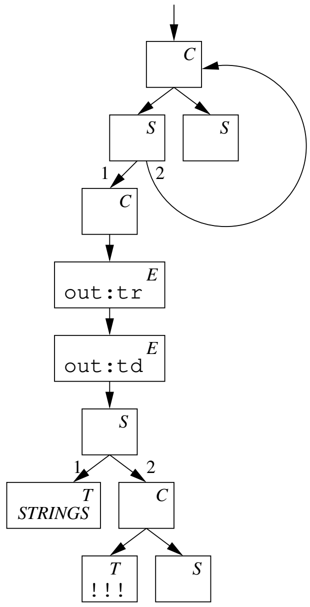


Figure 13: Completed summary graph fragment.

Sorting

If the `apply-templates` instruction contains any `sort` directive, then we cannot rely on the order from the input document. This means that the generated content model must be scrambled to describe any order of element (but inferred cardinalities are of course preserved).

Example

The summary graph for the running example is shown in Figure 14.

11 Summary Graph Validation

We rely on an existing algorithm [11] for checking that $\mathcal{L}(SG) \subseteq \mathcal{L}(D_{out})$. (The modifications to the definition of summary graphs as mentioned in Section 8 are easily incorporated in the implementation.)

A typical validity error report from the tool looks as follows:

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

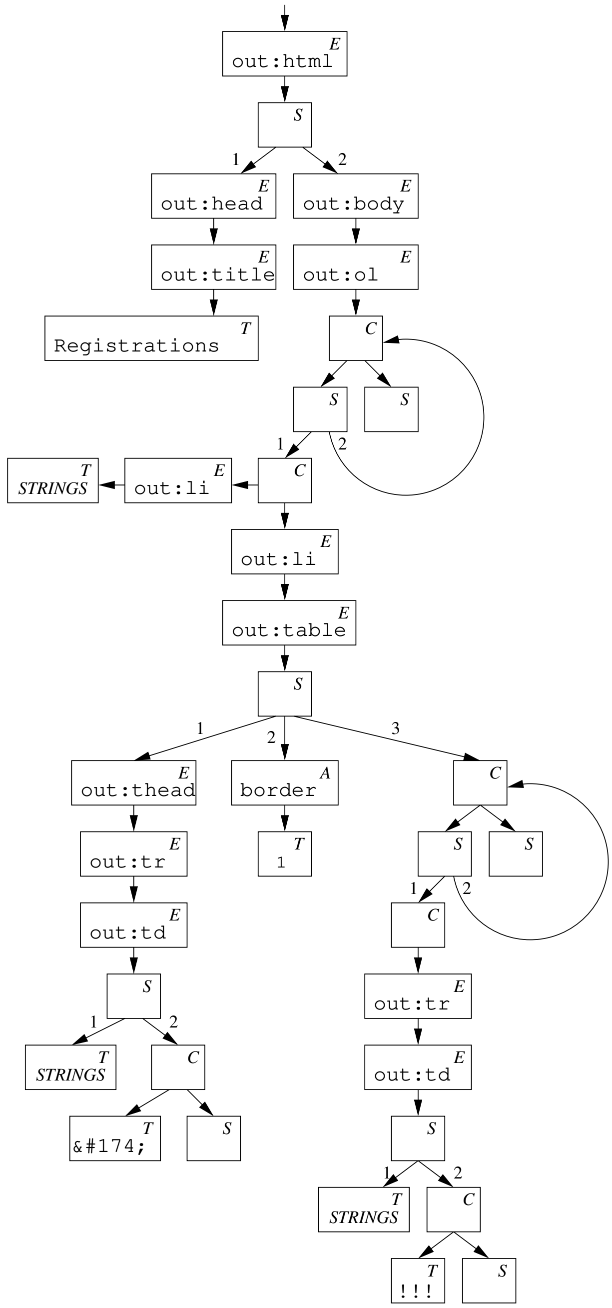


Figure 14: Summary graph for the example.

This report consists of four parts:

- The error message, describing the nature of the inconsistency with the output schema.
- The signature of the involved template rule.
- The context node that is used for instantiating the template when the inconsistency occurs.
- The relevant fragment of the output DSD schema showing the expected content model.

The summary graphs that are being produced contain information that allows detection of other problems besides validity errors, for example

- attempts to insert attributes in non-element nodes or non-text contents in attribute nodes;
- `select` expressions that never select anything or `match` expressions that never match anything (dead code); or
- insertion of attributes after children have been added to an element.

Our present implementation does not exploit these opportunities, but we plan to do so in future work.

12 Implementation and Experiments

We have developed a proof-of-concept implementation of our static validation algorithm. It consists of several components, some of which were available off-the-shelf. A DTD parser is available from `www.wutka.com`, an XML API from `www.jdom.org`, and an XPath parser from `www.jaxen.org`. DTD schemas are translated into DSD2 schemas [29], and the summary graph validation from Section 11 uses the previously published algorithm [11]. Thus, the novel components of our implementation are the simplifier from Section 6, the flow analysis from Section 7, the summary graph construction from Section 9, and a main part that combines the various components.

Test cases for our tool consist of triples of the form *(input DTD schema, XSLT stylesheet, output DTD schema)*. Such instances are remarkably difficult to obtain, since publicly available stylesheets often work on esoteric input languages for which no documentation is readily available. We have, however, collected 15 interesting triples of which two are written by ourselves (independently of this project). In some cases where we could only obtain a schema for either the input or the output language, we used the SAXON DTDGenerator [20] to create schemas from sample documents. At least for input schemas, this should be a safe approximation. Figure 15 shows our collection of benchmark triples, which is seen to contain stylesheets of small to medium sizes and

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

Figure 15: Benchmark triples, sizes in lines.

schemas ranging from small to largish. Often, the output language is XHTML and for some of these cases we choose to use directly the corresponding DSD2 schema, which is able to capture more requirements than a DTD schema.

The precision of our tool is presented in Figure 16 which classifies the generated error reports. True errors are those that may actually produce invalid output.

Encouragingly (for our tool, not for the stylesheet authors), a significant number of true errors were reported. They range over a number of different problems:

- misplaced elements, such as `link` elements occurring outside the XHTML header;
- undefined elements, attributes, or attribute values;
- missing elements or attributes, for XHTML typically the `alt` attribute of `img` elements or the `title` element in the header;
- unexpected empty content, for XHTML typically `ul` or `ol` lists that cannot be guaranteed to contain at least one `li` element; and
- wrong namespaces, which typically occurs when nodes are copied directly from input to output without realizing that the namespace must be changed.

Most errors are easily found and corrected, but in a few cases the intentions of the stylesheet author escape us. To illustrate the variety of errors found, we list

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

Figure 16: Results of static validation.

the first line of the six unique kinds of errors among the 12 error messages for `slideshow.xsl`:

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

These describe sloppy use of XHTML, but the resulting output would of course be accepted by most browsers. For non-XHTML applications, the consequences of such errors could be much worse.

The three false errors show cases where the approximations in our algorithm are too coarse:

- The two false errors in the validation of `CreateInvoice.xsl` both originate from instances where a `select` attribute has value of type `//foo`, which means "any `foo` element occurring in the document". However, it turns out that in this particular case, the `select` expressions could be simplified to just `foo`, in which case our current level of approximation is adequate.

- The false error in the validation of `ontopia2xtm.xsl` occurs when an attribute value is tokenized using the XPath `substring` function, and a template is instantiated for each token. Constructs like these are inherently difficult to analyze, but fortunately not common.

Stylesheet	FG	SG	Flow	Build	Analyze	Total
poem.xsl	26	95	0.22	0.07	0.05	0.93
AffordableSupplies.xsl	4	22	0.05	0.05	0.28	1.07
agenda.xsl	10	38	0.08	0.06	0.08	0.83
news.xsl	21	81	0.18	0.08	0.07	0.92
CreateInvoice.xsl	25	100	0.25	0.11	0.86	1.77
adressebog.xsl	33	412	0.19	0.20	0.32	1.32
order.xsl	31	173	0.26	0.11	0.16	1.17
slideshow.xsl	51	254	0.36	0.14	0.82	2.11
psicode-links.xsl	70	304	0.42	0.15	0.19	1.45
ontopia2xtm.xsl	82	318	0.34	0.20	0.83	2.08
proc-def.xsl	33	344	0.37	0.19	0.80	2.10
email_list.xsl	61	291	0.39	0.18	0.35	1.69
tip.xsl	113	492	0.69	0.24	0.28	1.92
window.xsl	100	515	0.41	1.47	3.02	5.83
dsd2-html.xsl	412	72,699	6.95	15.22	56.17	79.55

Figure 17: Performance for validating benchmark triples.

We have in addition considered the following generic identity transformation:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/|@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Our static validation algorithm has been designed to always handle such transformations correctly, and we have verified this property on a large selection of DTD schemas.

The performance of our tool is shown in Figure 17. Here, "FG" shows the combined number of nodes and edges in the constructed flow graph, "SG" is the combined number of nodes and edges in the constructed summary graph, "Flow" is the time to perform the flow analysis, "Build" is the time to construct the summary graph, "Analyze" is the time to analyze the inclusion into the output language, and "Total" is the time for running the entire tool (all measured in seconds). All experiments were performed on a 3GHz Pentium 4 with 1 GB RAM running Linux. The numbers are seen to be reasonable for these examples. Note that dsd2-html.xsl is larger and more complicated, which is quite obvious in the running times.

In Figure 18 we report the running times for the static validation of the identity transformation on a number of different DTD schemas. This clearly shows

Schema		SG	Flow	Build	Analyze	Total
`news.dtd`	12 lines	157	0.16	0.16	0.12	0.82
`dsd2.dtd`	104 lines	3,853	0.44	1.12	1.41	3.52
`xhtml.dtd`	1,198 lines	26,110	14.46	6.37	3.40	25.04
`fo.dtd`	1,480 lines	90,544	581.49	25.56	7.32	615.32

Figure 18: Performance for validating the identity transformation.

that the performance of the current implementation of our tool may not scale to seriously large instances. However, many of our low-level data structures and algorithms are currently rather naive, so there is ample basis for optimizations. Note that the time for `fo.dtd` seems disproportionately high. This is because we count the number of lines before expansion of entity references, while `fo.dtd` actually expands to more than 12,000 lines due to an extreme number of attribute definitions.

13 Conclusion

We have presented the first working tool that is capable of performing static validation of XSLT stylesheets. Based on a pragmatic approach that involves examination of hundreds of existing stylesheets, the technique we have developed is shown to handle typical cases with sufficient precision and performance to be practically useful.

The work presented here may be continued in various ways. First, it is possible to support XML Schema as a replacement for DTD. The ground work for this extension is described in [22]. Second, we believe that our approach can be generalized to also handle XSLT 2.0 although this involves a number of challenges for the flow analysis. Moreover, as mentioned in Section 11, our approach can also be used to detect other problems besides validity errors. In another direction, the information provided by our analysis may also be useful for optimizing code generation and for debugging purposes.

Acknowledgements

We thank Søren Kuula for his many detailed comments and valuable suggestions to this work.

References

[1] Altova. XMLSpy, 2005. `http://www.altova.com/xmlspy`.

[2] Jacek Ambroziak et al. XSLTC, 2004. `http://xml.apache.org/xalan-j/xsltc/xsltc_compiler.html`.

[3] Philippe Audebaud and Kristoffer Rose. Stylesheet validation. Technical Report RR2000-37, ENS-Lyon, November 2000.

[4] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. In *Proc. 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '05*, pages 25–36, 2005.

[5] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[6] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *Proc. 19th European Conference on Object-Oriented Programming, ECOOP '05*, volume 3586 of *LNCS*. Springer-Verlag, July 2005.

[7] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, pages 221–231, June 2001.

[8] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[9] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (third edition), February 2004. W3C Recommendation. `http://www.w3.org/TR/REC-xml`.

[10] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Static analysis for dynamic XML. Technical Report RS-02-24, BRICS, May 2002. Presented at Programming Language Technologies for XML, PLAN-X '02.

[11] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.

[12] James Clark. XSL transformations (XSLT), November 1999. W3C Recommendation. `http://www.w3.org/TR/xslt`.

[13] James Clark and Steve DeRose. XML path language, November 1999. W3C Recommendation. `http://www.w3.org/TR/xpath`.

[14] Ce Dong and James Bailey. Static analysis of XSLT programs. In *Proc. 15th Australasian Database Conference, ADC '04*. Australian Computer Society, January 2004.

[15] Denise Draper et al. XQuery 1.0 and XPath 2.0 formal semantics, November 2002. W3C Working Draft. `http://www.w3.org/TR/query-semantics/`.

[16] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML processing in Java. In *Proc. 14th International Conference on World Wide Web, WWW '05*, pages 278–287. ACM, May 2005.

[17] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, April 1979.

[18] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[19] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95*, pages 393–407, January 1995.

[20] Michael Kay. Saxon, 2004. `http://saxon.sourceforge.net/`.

[21] Stephan Kepser. A proof of the Turing-completeness of XSLT and XQuery. Technical report, SFB 441, University of Tübingen, 2002.

[22] Christian Kirkegaard and Anders Møller. Type checking with XML Schema in Xact. Technical Report RS-05-31, BRICS, 2005.

[23] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[24] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1.

[25] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. World Scientific Publishing Company.

[26] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '93*, January 1993.

[27] Wim Martens and Frank Neven. Typechecking top-down uniform unranked tree transducers. In *9th International Conference on Database Theory*, volume 2572 of *LNCS*. Springer-Verlag, January 2003.

[28] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66:66–97, February 2002.

[29] Anders Møller. Document Structure Description 2.0, December 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from http://www.brics.dk/DSD/.

[30] Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. Tenth International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[31] Chimezie Ogbuji. Visualizing XSLT in SVG, 2003. http://www.xml.com/pub/a/2003/06/04/xslt-svg.html.

[32] Ovidiu Predescu and Tony Addyman. XSLT-process, 2005. http://xslt-process.sourceforge.net/.

[33] Thomas Schwentick. XPath query containment. *ACM SIGMOD Record*, 33(1):101–109, 2004.

[34] Stylus Studio. XSL Debugger, 2005. http://www.stylusstudio.com/xsl_debugger.html.

[35] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema part 1: Structures second edition, October 2004. W3C Recommendation. http://www.w3.org/TR/xmlschema-1/.

[36] Akihiko Tozawa. Towards static type checking for XSLT. In *Proc. ACM Symposium on Document Engineering, DocEng '01*, November 2001.

[37] Philip Wadler. A formal semantics of patterns in XSLT and XPath. *Markup Languages*, 2(2):183–202, 2000.

[38] Peter T. Wood. Containment for XPath fragments under DTD constraints. In *Proc. 9th International Conference on Database Theory, ICDT '03*, pages 300–314. Springer-Verlag, 2003.

Recent BRICS Report Series Publications

RS-05-32 Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. ***Static Validation of XSL Transformations*****. October 2005. 50 pp.**

RS-05-31 Christian Kirkegaard and Anders Møller. ***Type Checking with XML Schema in*** **XACT. September 2005. 21 pp.**

RS-05-30 Karl Krukow. ***An Operational Semantics for Trust Policies*****. September 2005.**

RS-05-29 Olivier Danvy and Henning Korsholm Rohde. ***On Obtaining the Boyer-Moore String-Matching Algorithm by Partial Evaluation*****. September 2005. ii+9 pp. To appear in** ***Information Processing Letters*****. This version supersedes BRICS RS-05-14.**

RS-05-28 Jiří Srba. ***On Counting the Number of Consistent Genotype Assignments for Pedigrees*****. September 2005. 15 pp. To appear in FSTTCS '05.**

RS-05-27 Pascal Zimmer. ***A Calculus for Context-Awareness*****. August 2005. 21 pp.**

RS-05-26 Henning Korsholm Rohde. ***Measuring the Propagation of Information in Partial Evaluation*****. August 2005. 39 pp.**

RS-05-25 Dariusz Biernacki and Olivier Danvy. ***A Simple Proof of a Folklore Theorem about Delimited Control*****. August 2005. ii+11 pp. To appear in** ***Journal of Functional Programming*****. This version supersedes BRICS RS-05-10.**

RS-05-24 Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. ***An Operational Foundation for Delimited Continuations in the CPS Hierarchy*****. August 2005. iv+43 pp. To appear in the journal** ***Logical Methods in Computer Science*****. This version supersedes BRICS RS-05-11.**

RS-05-23 Karl Krukow, Mogens Nielsen, and Vladimiro Sassone. ***A Framework for Concrete Reputation-Systems*****. July 2005. 48 pp. This is an extended version of a paper to be presented at ACM CCS'05.**

RS-05-22 Małgorzata Biernacka and Olivier Danvy. ***A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines*****. July 2005. iv+39 pp.**

RS-05-21 Philipp Gerhardy and Ulrich Kohlenbach. ***General Logical Metatheorems for Functional Analysis*****. July 2005. 65 pp.**