Master's Thesis

Static Validation of XSLT

Mads Kristian Østerby Olesen, madman@daimi.au.dk

Advisors:
Anders Møller, amoeller@brics.dk
Michael I. Schwartzbach, mis@brics.dk

June 1, 2004

Abstract

XML has become a widely popular standard for storing and exchanging structured data, particularly on the Web. Many languages and language extensions have been proposed for manipulating XML data, and a number of these employ static analyses in order to statically guarantee the validity of their XML output against some schema.

Among the XML manipulation languages, XSLT has been embraced as an XML-to-XML tranformation language, and although some research efforts have gone into statically analyzing the output of such transformations, no pratically useful analysis technique has so far come out of it. We attempt to remedy this by designing such an analysis technique, conservatively approximating an answer for the property of schema-conforming output.

Our analysis technique centers around a detailed analysis of the possible flow of template instantiations in the transformation, as well as the application of an abstraction, which has already proven useful in other contexts, for modelling the construction of XML values.

Empirical study shows the analysis to perform well enough for practical use, and errors are found in a number of transformations gathered from various independent sources.

Table of Contents

1 Introduction **1**
1.1 Setting the Scene . . . 1
1.2 Summary of Contributions . . . 2
1.3 Overview of Chapters . . . 2

2 Background **3**
2.1 Markup Languages . . . 3
2.2 Extensible Markup Language (XML) . . . 4
2.3 Classes of XML Documents . . . 7
2.4 Related XML Standards . . . 9
2.5 The XPath Data Model . . . 11
2.6 XML Path Language (XPath) . . . 12
2.7 XSL Transformations (XSLT) . . . 15
2.8 Summary . . . 19

3 Static Output Validation of XSLT **21**
3.1 Motivation . . . 21
3.2 The Problem . . . 23
3.3 Related Work . . . 27
3.4 Goals and Restrictions . . . 33
3.5 Summary . . . 34

4 Analysis and Design **35**
4.1 The Analysis Approach . . . 35
4.2 Data Mining . . . 37
4.3 Overview . . . 39
4.4 Simplification . . . 40
4.5 The Flow Analysis . . . 46
4.5.1 The XSLT Flow Graph . . . 53
4.5.2 The XSLT Flow Analysis . . . 58
4.5.3 Path Simulation Test . . . 64
4.5.4 Paths Automata Test . . . 67
4.5.5 The Priority Override Filter . . . 73
4.5.6 The Result . . . 76
4.6 Parameter Analysis . . . 76
4.7 The Summary Graph Analysis . . . 78

4.7.1 Summary Graph Construction 80
4.7.2 Summary Graph Inclusion Analysis 87
4.8 Summary 88

5 Experimentation and Interpretation 89
5.1 Implementation Details 89
5.2 The Examples 94
5.3 Precision 96
5.4 Performance 101
5.5 Summary 104

6 Conclusions 105
6.1 Evaluation 105
6.2 Contributions 107
6.3 Future Work 108

A Appendix 111
A.1 Output Summary Graph for the News Transformation 112
A.2 Flow Graph for General Identity on the News DTD 113
A.3 A Fragment of the ontopia2xtm.xsl Example 114

References 116

List of Figures

2.1 Example XML Document 5
2.2 Example DTD 6
2.3 XPath Step Evaluation 13
2.4 Example XSLT Transformation 17

3.1 The General Identity 26

4.1 XSLT Transformation Sizes 37
4.2 XSLT Construct Mining 38
4.3 Select and Match Mining 39
4.4 Overview of the Analysis 39
4.5 Reduced XSLT Grammar 45
4.6 The Simplified News Transformation 47
4.7 Imprecise General Identity Flow Graph 49
4.8 Precise General Identity Flow Graph 49
4.9 Flow Graph for the News Example 54
4.10 Example Run of the Location Path Simulation 67
4.11 Example Run of the Path Expression Construction 71
4.12 Paths Automaton for the News DTD 72
4.13 Summary Graph Template Grammar 79
4.14 Example Summary Graph 80
4.15 An Unfolding of the Example Summary Graph 80
4.16 Example Content Model Fragment 83
4.17 Single Step Selection Fragments 85
4.18 Multiple Step Selection Fragments 86

1

Introduction

1.1 Setting the Scene

The Extensible Markup Language (XML) [18] of the World Wide Web Consortium (W3C) has gained much popularity as a standard for describing and exchanging structured data, particularly on the Web. Today, one of the great strengths of XML is precisely this popularity. A large number of languages and tools have been developed and are available, freely or commercially, to developers using XML. This is a self-increasing effect, and XML seems to be here to stay.

A great focus on XML development is the use of *XML schemas* for describing domain-specific languages obeying the XML syntax. These schemas provide safety by allowing one to check—in a standardized way—whether some XML document is of the expected format. This is commonly used to ensure correct input to programs. The formal character of these schemas also gives rise to the possibility of static analysis of XML manipulation programs, i.e. analyzing statically the types of XML values in programs, as well as the output of these programs.

One widely accepted XML manipulation language is XSL Transformations (XSLT) [23], which in essence is a language for transforming an XML document into another XML document through structural recursion. Many efforts have gone into trying to statically determine the validity of XSLT transformation output, but no practically useful tool has yet been developed. The goal of this thesis is to lay the foundations for such a tool by designing a sufficiently fast and accurate static analysis technique, which is able to assist in the debugging of XSLT transformations and which allows

for static guarantees to be issued on the output of these transformations.

1.2 Summary of Contributions

The contributions of this thesis can be summarized as follows:

- A static analysis technique for determining whether all output of an XSLT transformation conforms to a specified DTD.
- A flow analysis technique exposing the structure in which templates are instantiated in an XSLT transformation. This shall be a vital tool in our static analysis technique.
  The core of the flow analysis is a couple of techniques for determining whether two XPath location paths can accept the same kinds of nodes, as well as a test for statically resolving priorities between the XSLT template rules.
- An alternate form of the *summary graph* abstraction with accompanying schema validation algorithm, which has proven useful in other contexts of XML manipulation. This model shall be used for describing the possible outputs of an XSLT transformation.
- An implementation of our static analysis technique, successfully analyzing and finding errors on XSLT transformations written independently of this project. Experimental evidence, indicating the practical usefulness of our analysis technique in (1) locating errors in transformations, and (2) establishing static guarantees of output validity.
- The outline of a simplification technique for reducing XSLT transformations to a more manageable form, thereby easing the job for our analysis.

1.3 Overview of Chapters

In Chapter 2 we shall examine the details of XML and a number of its related standards, including XSLT. In Chapter 3 we describe more precisely the problem we wish to solve, and we survey related work in the field. Chapter 4 describes in full detail the static analysis approach we employ, while Chapter 5 describes our implementation and contains a series of experiments trying to determine the usefulness of our approach, and whether we achieved our goals. Finally, in Chapter 6 we round up by evaluating our work and examining further work that would be natural to do in extension to the work done in this thesis.

2

Background

This chapter presents the basic knowledge and terminology upon which the thesis rests. The content presented here will be the frame of reference for the remaining chapters.

2.1 Markup Languages

The term “markup” refers to the traditional publishing practice of “marking up” manuscripts that are to be typeset. More recently, the word also refers generally to marking up the structure of a document, and not just to formatting.

The emergence of *Standard Generalized Markup Language (SGML)* [74] was an attempt to formalize the specification of markup languages on computers. SGML consists of a standardized way of marking up documents through *markup tags*, and a *meta-language* for specifying the syntax of markup languages.

SGML ended up too complicated to be widely adopted. One SGML application did however catch the eye of the public: the *Hypertext Markup Language (HTML)* [44] by Tim Berners-Lee and Robert Caillau.

HTML was originally designed for formatting research documents at CERN, but has since evolved into a rather complex formatting and layout language. It is presently the dominating standard for presenting information on the Web, and is probably the most wide-spread markup language to date. The latest version is HTML 4.01 [70], developed by the World Wide Web

Consortium (W3C).

Other examples of SGML applications include: TEI [76], DocBook [63], and HyTime [73].

2.2 Extensible Markup Language (XML)

The increased importance of data exchange on the web, particularly in the relatively new field of application-to-application Web services, where both server and client is an application, has given rise to the *Extensible Markup Language (XML)* [18] project of the *World Wide Web Consortium (W3C)*. XML is derived from SGML and HTML, recognizing the problematic nature of the complexity in SGML and simplifying it. Certainly, SGML is more customizable than XML, but also much more expensive to implement.

XML provides:

- A platform independent standard for storing and exchanging structured data.
- The ability to define markup languages tailored to each domain.
- A large base of existing technologies ready for use, including parsing, manipulation and validation tools.

XML is widely used as a data storage and exchange format in the information technology business today. Since XML is rather central to this thesis, we shall examine it in some detail in the following.

Basically, XML is just a linear syntax for unranked, ordered and labeled trees. *Elements* form the structure of the tree and are specified by start- and end-tags such as `<mytag>...</mytag>`, enclosing further elements, plain text, comments and other content. Elements can be annotated with name-value pairs called *attributes* written `myattribute="somevalue"`. In their basic and most used form the value is simply a string. These *StringType attributes* can be considered nodes in the tree having a single text leaf. However, in contrast to the elements and other content, attributes are unordered. An example of an XML document can be found in Figure 2.1, describing news items.

Apart from StringType attributes, there are a number of other forms of attributes, most notably the ID, IDREF, and IDREFS of the *TokenizedType attributes*. They are used to assign unique string identifiers to elements and referring to the elements through these identifiers. Also, the *EnumeratedType attributes* only allow one of a specified set of values.

IDs and IDREFS can be interpreted as node-to-node references on level with parent-child relations, as is done in XML-QL [31]. In this view, XML documents become general unordered graphs.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<news>
  <item category="international" date="May 7, 2004" time="10:30 PM">
    <headline>Rumsfeld apologizes</headline>
    <text>
      <p>Defense minister Donald Rumsfeld apologizes for the
      mistreatment of iraqi prisoners.</p>
      <p>Constant inquisitive questioning by democratic and republican
      senators made it clear that the apology, in their opinion, was
      too late.</p>
    </text>
  </item>

  <item category="national" date="May 6, 2004" time="3:34 PM">
    <headline>Carlsberg still on to China</headline>
    <text>
      <p>Carlsberg is still interested in the Chinese market, despite
      that competition in the market is tight.</p>
    </text>
  </item>
</news>
```

Figure 2.1: An example of an XML document, in this case representing news items.

Processing instructions are a means to pass instructions to applications. They are not intended as part of the actual data. The format of each processing instruction is entirely up to their designers, so long as they fit into the `<?pi_identifier ... ?>` construct.

General entities and *Parameter entities* allow re-use and recursive declaration of fragments of content in XML documents. Basically, they are named declarations of content, and the two occupy distinct namespaces. Parameter entities are resolved before general entities, and are used exclusively in the declaration part of the document. Thus only general entities may be referenced from the actual XML content. *External general and parameter entities* can be used to get data from the environment, which is not necessarily in the same character encoding, and not necessarily text at all.

All XML documents must be *well-formed*, which entails conforming to the grammatic productions and *well-formedness contraints* of the specification. As an example, the *Element Type Match* constraint requires that the name in an element's end-tag matches the name in its start-tag[1].

XML, having been designed for the Web, necessitates platform independence, which has been achieved by allowing a variety of character encodings. The character encoding used in some document is specified in the document *prolog*. Although providing platform independence, this feature at the same

[1]The term well-formed XML is often used to refer to the Element Type Match constraint alone, but it does in fact cover much more. See the XML specification [18] for further details.

```
<!ELEMENT news (item*)>

<!ELEMENT item (headline,text)>
<!ATTLIST item category (national|international|business)  "national"
               date CDATA #REQUIRED
               time CDATA #REQUIRED>

<!ELEMENT headline (#PCDATA)>

<!ELEMENT text (p+)>

<!ELEMENT p (#PCDATA)>
```

Figure 2.2: An example DTD, describing a syntax for representing news items.

time complicates the task of writing a parser for XML.

Typically, an application of XML involves a more or less fixed "notation" for describing data in a certain domain. This notation defines a *class of XML documents* - also called an *XML markup language*. An important part of XML is the *Document Type Declaration (DTD)*, which allows one to formalize the description of such a class of documents. In this respect DTD is a metalanguage for specifying the grammar[2] of markup languages.

The grammar consists of a set of *element content models* and *attribute lists*. The element content models describe the element vocabulary and the legal sub-content of each type of element—including order—through *deterministic regular expressions*, as well as a further restricted model for content, including plain text. The attribute lists define possible attributes and their types for each element name. Figure 2.2 shows a DTD for news items. The example XML document of Figure 2.1 conforms to this DTD.

The DTD can be internal or external to the XML document, or both. The DTD is also where entities are declared, and the DTD can be recursively constructed through the use of entities like the rest of the document. However, unlike its predecessor SGML, XML allows "undeclared" markup—i.e. markup with no attached DTD.

The XML specification defines an XML document as *valid*, if it refers to a DTD and conforms to the rules described therein.

Although DTDs undoubtedly are useful, they have proven too weak in practice to describe all desirable markup languages. This has lead to further research on the subject. More about this and XML languages in general in Section 2.3.

Today, the term XML commonly refers to the entire *W3C XML Architecture*, which is a family of XML related standards by the W3C. These standards will be described in more detail in Section 2.4.

[2]Grammar as in the general sense: A set of rules describing syntax.

2.3 Classes of XML Documents

As described in the previous section, DTDs can be used to formalize XML markup languages. This formalization brings some palpable benefits:

- It provides a precise reference for the developers involved as well an easily readable definition for newcomers to the language.
- It allows documents to be validated against the DTD automatically by a progam.
- The automatic validation implies that applications can assume valid input, thereby reducing their complexity.

But as also mentioned in the previous section, the DTD language has proven too weak in practice. For example, an element of a certain name will always be subject to the same constraints no matter the context it appears in, i.e. the content model can not depend on for example which type of parent the element has. Also, DTDs are unable to describe what root element or elements are required of documents conforming to the DTD, although this is often desirable. Several alternative languages have been suggested to try to remedy the many shortcomings of DTDs. Unfortunately no consensus has been reached.

Languages such as DTD are called *XML schema languages*[3]. In general schema languages specify grammars for describing classes of XML documents. An XML document conforming to some schema is called an *instance document* of that schema. Adopting the vocabulary of the XML specification, we shall say that a document instance is *valid* if it conforms to the constraints expressed in the schema. Thus, a *schema processor* or *validating parser* checks the input document against the schema constraints and reports any errors. After parsing, the document is returned in *normalized* form, where for example default attribute values have been inserted, and entities resolved.

When XML was designed, the authors did not anticipate the inadequacy of the DTDs. As a consequence, XML 1.0 incapable of pointing to any types of schemas other than DTDs. References to schemas written in other languages must be referred to in other ways. For example, the XML Schema language achieves this through a global attribute: `schemaLocation` in the XML Schema namespace (More on global attributes in Section 2.4).

A consequence of the lack of consensus on which schema language to use, is that you cannot expect every tool you encounter to understand the specific

[3]The term *schema* is a greek word, among other things meaning “A diagrammatic representation”, or ”an outline or model”.

schema language you have employed. This leaves room for DTDs to act as a sort of “common language”.

Certainly, DTD—being part of the XML specification—is the most widely used schema language today. A DTD is often used as a “fallback” schema, combined with a more powerful schema language to describe the exact XML language. Since most people involved in XML development know DTDs, the DTD fallback allows people, who are not familiar with the specific schema used, to understand the basics of the XML language. Similarly any application working with XML documents and their classes can be supplied with the fallback, in case the application does not know the original schema language. Of course, this may result in less than perfect—but hopefully better than nothing—results.

One important alternative to DTDs is the *XML Schema*[4] [78] language. It is designed by the W3C (same as XML), and is their attempt to cover the shortages of the DTDs. Certainly XML Schema belongs to a more expressive category than DTDs [53], and other important improvements have been made in the language, such as supporting XML Namespaces [16] (see Section 2.4). But the language still leaves room for improvement. For example, the inability of DTD to specify a root element, as well as its inability to express context dependency, is still present in XML Schema. And for some reason XML Schema ended up being very complex. The specification is very hard to comprehend, and the schemas made with it also tend to be tough to read.

Fortunately it seems that it is possible to have the expressiveness of XML Schema without its complexity. The *Document Structure Description 2.0 (DSD2)* [60] does exactly that. The specification is remarkably simple compared to that of XML Schema.

Many other schema languages have been proposed: XML-Data [52], DCD [15], DDML [11], SOX [29], Assertion Grammars [69], Schematron [46], TREX [24], Examplotron [80], RELAX NG [26], and more.

Classes of XML documents are often referred to as *XML types*, since the schemas restrict XML documents in the same manner as types do variables in general-purpose languages. We shall refer to them mostly as *XML classes* to avoid any ambiguities in connection with type systems.

The most widely known XML language is XHTML [65], which is a reformulation of HTML 4.01 [70] into XML. Other examples of XML languages are: CML [61], WML [82], ThML [22], XSLT [23], VoiceXML [12], SVG [37]. We shall examine the XSLT language in much greater detail in Section 2.7

[4]Note that “XML Schema” with a capital S refers to the XML Schema standard, while “XML schema” refers simply a schema for XML in general.

2.4 Related XML Standards

Over the years, the core XML specification has been augmented and built upon with a number of related technologies. We will here examine some of the most central: XML Namespaces [16], XPointer [39], XInclude [56], and XLink [30]. The XPath [25] and XSL [1] standards will in later sections be examined for their special relevance to this thesis. There are many more standards in what is called the *XML Architecture*. See "W3C Technical Reports and Publications" [84] for an overview of these.

XML Namespaces

One of the most important—and perhaps *the* most important—W3C standard from an XML viewpoint is XML Namespaces [16]. The standard is an augmentation of the original XML standard, and it refines the basic syntax of XML documents. The standard has quickly been embraced, and is often considered to be the new "core XML standard".

The idea behind XML Namespaces is that with the many different classes of XML documents, it is natural to start mixing them, i.e. having documents containing data of several different XML classes intermixed. For example to contain formatted text as XHTML, instead of having to design a whole new set of formatting elements. If the classes are nicely separated in the document there may not be any ambiguities, but in a setting of complex intermingling of elements and where some element names are shared between the classes it is a different matter.

To solve this, W3C came up with the XML Namespaces extension of the XML standard. It basically requires names (element names, attribute names and the like) to be paired with a Uniform Resource Identifier (URI), uniquely identifying the XML class—or more precisely the *namespace*[5]—from which it originates. Writing these URIs at each name would of course seriously impede an already verbose syntax, so the standard allows one to define *prefixes* referring to these URIs. A prefix is a sort of alias for a namespace URI, and the prefix name is local to the document. Furthermore, as documents usually contain a greater part of its content in a single XML class, the *default namespace* allows one to omit prefixing elements of this dominant class.

A point to note is the *namespace partitions* which are part of the XML namespaces standard. There are three major partitionings of the names in an XML class, within which the names must be unique:

- **Element names** have a partition.
- **Local attributes** have a partition for each element type.

[5] An XML namespace URI actually refers to several namespaces in the form of the *namespace partitions*.

- **Global attributes** have a partition.

So to clear it up: element names naturally reside in a single namespace, as they must be unique among themselves. Different elements can without loss have attributes with the same names, because the attributes are local to the element. Therefore the attributes local to an element type have their own namespace: one namespace per element type. Finally, the concept of *global attributes* refers to attributes which are not bound to a single element. Global attributes must be prefixed while locals must not. Globals can therefore not collide with the local attribute names. Defining global attributes and constraining their occurrence is for the schema to describe.

A final detail to consider is that the DTDs of the core XML standard are completely ignorant of XML Namespaces. For namespaces support, one must seek other schema languages such as XML Schema or DSD2.

XPointer, XInclude, and XLink

The XPointer standard forms the base of XInclude and of XLink. It is essentially a syntax for pointing to XML resources. Both whole documents and fragments of them. XPointer extends XPath, the basic XML node-selection language, with a few concepts—most notably *points* and *ranges* for more precise pointing—and defines a suitable syntax for specifying the so-called *XPointers*.

XInclude provides a syntax for modularizing XML documents through the use of URIs and XPointer, by specifying a syntax for inserting XML documents or fragments thereof at specified locations. Also some forms of processing can be applied to documents, such as converting to text and translating between UNICODE encodings.

XLink[6] on the other hand is an XML notation that very generally allows one to describe relationsships between resources. It is basically a more general and powerful version of the well-known HTML hyperlinks. XPointer may be used to refer more precisely to XML fragments, if such is needed.

Work In Progress

One must be aware that the W3C XML standards are in constant evolution. This means that work done in this field must be continuously updated or fall behind. As an example, the XML 1.0 core standard [18] was, in February 2004, upgraded from second to third edition. The changes were mostly small details regarding UNICODE [77] support. Also released in February 2004 as a W3C Recommendation was XML 1.1 [19][17] (also mostly dealing with UNICODE issues), and XHTML 2.0 [6] is in the works as a Working Draft, and has been for some time. Many more standards have been produced by

[6] Formerly known as the Extensible Linking Language (XLL).

the W3C or are in progress. The reader is again referred to "W3C Technical Reports and Publications" [84] for a thorough listing of their work.

2.5 The XPath Data Model

As mentioned in Section 2.2, XML is simply a linear syntax for unranked, ordered and labeled trees. Behind this short statement however is a perhaps slightly more complex logical model. This model can and has been expressed in various ways. The W3C specifications for DOM [83], XPath [25], and XML Information Set [28] each define such a logical model, and none of them quite agree.

Nonetheless, it is advantageous to view XML through the eyes of a specific logical model, and for the topic of this thesis, the *XPath data model* [25] is the most relevant. So in the interest of uniformity, the XPath data model will be the frame of reference regarding XML structure throughout the thesis. This model is the foundation of the XPath location paths, described in detail in Section 2.6. Note that the XPath data model's relation to the XML Information Set in particular is described in an appendix of the XPath specification.

The XPath data model defines a set of *node types*, which covers all the different entities found in an XML document. The node types are related to each other in a tree structure. The node types of the data model are:

- Root nodes
- Element nodes
- Text nodes
- Attribute nodes
- Namespace nodes
- Processing instruction nodes
- Comment nodes

The *element nodes* again form the core of the structure by being the only type of node—apart from the root node—that can have children. Each element is labeled, and can have any number of children, attributes and namespace nodes. Thus the attribute and namespace nodes are not considered children of an element, but they are "attached" to one, and that element *is* considered their parent element.

An *attribute node* has a name and a string-value[7], while a *namespace node*—not to be confused with namespaces themselves—is a prefix and URI pair corresponding to a namespace prefix declaration as defined in XML

[7]The string-value being the *normalized value* defined in the XML specification.

Namespaces. The namespace prefix declarations themselves—although being attributes—are replaced by the namespace nodes. An element will have an attached namespace node for each namespace prefix in scope[8] at that element.

In order for the logical structure to be unambiguous, character data is grouped in as large chunks as possible into *text nodes*. As a consequence, text nodes never have other text nodes as preceding or following siblings.

The element nodes form an unranked tree, and the root of this tree is denoted the *document element*. As the name implies, the document element is an element node. The *root node* is the absolute topmost node in the node structure. It has as its only children the document element, along with any processing instruction or comment nodes that occur outside the document element tags in the XML document. The root node is *not* an element. It has no name, no attributes, and no namespaces. It serves purely as a "meta-root", and is easily confused with the document element. While the document element has a concrete representation in the XML document, the root node is purely a logical entity.

XML being ordered implies that the order of child nodes must be important in the data model as well. The XPath data model handles this through the concept of *document order*, which is basically the order in which the nodes are first encountered in the linear form of the document after expansion of general entities. In other words, elements are ordered according to the order of occurence of their start-tags. The document order is equivalent to the visiting order in a pre-order traversal. Attribute and namespace nodes occur after their parent element but before the children, and namespace nodes are defined to occur before attribute nodes. The *reverse document order* is the inverse of the document order.

Although the document order seems unexpectedly to impose an ordering on attribute and namespace nodes, it should be made clear that their ordering is implementation specific and should not be relied upon.

2.6 XML Path Language (XPath)

The *XML Path Language (XPath)* is a language for "selecting" or "pointing out" parts of XML documents. It may be used in many contexts. The best known today is probably XPointer, which is again used by XLink and XInclude to refer to fragments of other documents. XPath is the primitive that allows pointing to the internals of documents, and not just their wholes.

XPath is restricted to operate on XML Namespaces compliant documents, and of course operates under the XPath data model. The core of

[8] Namespace declarations in XML Namespaces are inherited to child elements of the containing element. Thus a prefix declaration is in scope for all element descendants unless "overridden" by a prefix declaration with the same prefix on such a child.

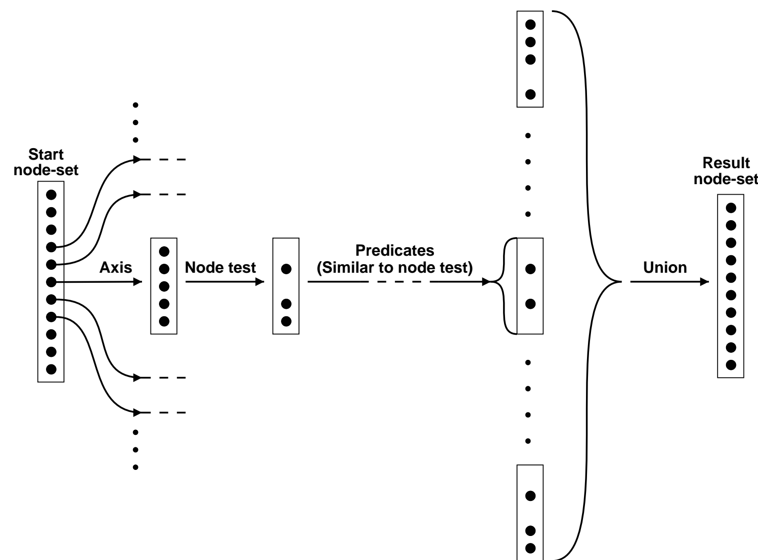


Figure 2.3: XPath evaluation model for a single *location step*.

XPath expressions are the *location paths*, used for selecting nodes in XML data.

A location path consist of a series of *location steps*, each consecutively producing a set of nodes working from the set produced by the previous step. The final *node-set*—i.e. the result of the last step—is the result of the location path evaluation. A node-set is one of the four types of values in the XPath language, and it is simply an unordered collection of references to nodes in some XML document[9]. See Figure 2.3 for a visualization of the evaluation model for one location step.

A location step is comprised of three parts:

- An *axis*, doing an initial rough selection of nodes by their structural relationships to the *context node* (the nodes in the source set). An example is the *child* axis, selecting all children of the context node.

- A *node test*, refining what kind of nodes we are interested in. This is often a specific element type such as `chapter`, leaving only elements with the name `chapter` in the set.

- Any number of *predicates*, which are essentially boolean expressions that filter out all nodes for which the expression is not true.

[9]Although the node-sets themselves are unordered, the document order implicitly imposes an ordering on the nodes in the set.

The XPath axes are: `child`, `attribute`, `namespace`, `descendant`, `self`, `descendant-or-self`, `parent`, `ancestor`, `ancestor-or-self`, `preceding`, `following`, `preceding-sibling` and `following-sibling`. Each axis is either a *forward- or reverse axis*. From the specification:

> "An axis that only ever contains the context node or nodes that are after the context node in document order is a forward axis. An axis that only ever contains the context node or nodes that are before the context node in document order is a reverse axis."

Thus the reverse axes are: `parent`, `ancestor`, `ancestor-or-self`, `preceding` and `preceding-sibling`. The `self` axis can be said to be "directionless", as it falls into both categories, but since it always selects a single node, the ordering makes no difference. The same can be said about the `parent` axis: it only ever selects one node or—for the root node—none at all.

Also, every axis has a *principal node type*, which is element nodes for all the axes that can select elements. This means most of the axes, except for the `attribute` and `namespace` axes, for which the principal node type is attribute and namespace nodes respectively.

The principal node type is of importance only in the *qname* and `*` node tests. The former accepts nodes of the given name and principal node type. The latter accepts simply all nodes of the principal node type.

XPath includes a full expression language with booleans, numbers and strings, as well as the *node-sets* which the location paths yield. These expressions allow quite complex filtering in the predicates. A common predicate is testing the *proximity position* of nodes through the `position()` function, for example to select every other node of the given type. For a forward axis, the proximity position will be according to the document order. For a reverse axis it will be the reverse document order.

The full syntax for XPath is somewhat cumbersome, so some syntactic sugar has been introduced. For example, if no axis is specified, the *child* axis is used as default. This—together with a couple of other shorthand notations—leads to a syntax very similar to directory paths of modern operating systems, so long as you restrict yourself to only the child axis. Certainly, the child axis is the most used of the axes.

For example, with the root node as context node, the location path `"news/item[@date]"` selects all the `news` elements with a `date` attribute (which under the news DTD of Figure 2.2 will always be present). `"news/item/*"` will select all `headline` and `text` elements. These two are *relative paths*, as the depend on a context node given before hand. This context node is provided by the parent language in which XPath is used, for example by XPointer. An *absolute path* on the other hand always starts from the root node, such as: `"/news/item/*"`.

A number of other things are provided by the parent language, such as *variable declarations* for variable resolution, and a *function library*. Xpath is a complex language, and for further details, we refer to the XPath specification [25].

A tricky couple of examples from the XPath specification, which are worth noting, are the expressions `"//para[1]"` and `"/descendant::para[1]"`. Although they might seem equivalent at first sight, they do in fact select quite different nodes. The former selects all `para` elements that are the first `para` child of their parent (note that `"//para[1]"` is short for `"/descendant-or-self::node()/child::para[position()=1]"`). The latter simply selects the first `para` element in the document. The difference comes from the way predicates filter the nodes: A predicate filters only the result of axis and node test operations *from a single node*. Thus, the *context position*, which is the one tested with the `"[1]"` predicate, is the position in this small intermediate set, not the position in the union of the sets. We again refer to Figure 2.3.

2.7 XSL Transformations (XSLT)

The *Extensible Stylesheet Language (XSL)* [1] standard has evolved from the much used HTML styling language *Cascading Style Sheets, level 1 (CSS1)* [54] and its successor *Cascading Style Sheets, level 2 (CSS2)* [10]. Further inspiration no doubt originated from the *Document Style Semantics and Specification Language (DSSSL)* [75] of SGML, which included a transformation language as part of the standard. An idea which was adopted by XSL, as we shall see.

Like its predecessors, XSL is designed to extract presentation specifics from data. This is in XSL achieved through two sub-standards[10]: *XSL Formatting Objects (XSL-FO)* and *XSL Transformations (XSLT)* [23]. XSL-FO is an XML language for describing precise layout and formatting, and XSLT is for transforming the data into a formatted document ready for presentation.

As it stands, although XSL evolved partly from CSS, CSS continues to be used widely, and it is still developed upon. At the time of writing CSS 3.0 is in development. It seems the foundation of existence for CSS is the fact that XHTML has not really taken over from HTML (and perhaps never will). XSLT has been designed so that it can output HTML but not use it as input, thereby rendering XSL unable to style HTML. Thus, the XSL and CSS working groups are eagerly developing on both styling systems.

[10]XPath is sometimes mentioned as a third sub-standard, although it was just as much designed for XPointer as well.

No matter what styling language one might prefer, XSLT—the transformation part of XSL—has its uses outside the styling community. It has proven a quite versatile and expressive XML transformation language, and has seen extensive use on the Web today. Huge amounts of transformations can be digged out of the web by the keen searcher.

XSLT has three output modes: XML, HTML and plain text, and although the text mode can be used to produce documents of whatever language you desire, XSLT was clearly developed with XML-to-XML transformations in mind. This is also how it is most used on the Web, and will be the focus in this thesis.

XSLT is a declarative language in XML format. An XSLT transformation is based on a number of *template rules*. A template rule is a *pattern* and *template* pair. The pattern is an XPath expression limited to a subset of XPath, and is a description of the kind of context this template rule can be applied to. The template is basically a fragment of the result tree, containing XSLT *instructions* which are used among other things for flow-control, complex data construction and invoking other template rules. The act of *executing* these instructions and producing a literal result tree fragment is called *instantiating* the rule or template. An example of an XSLT transformation can be found in Figure 2.4.

The core of the processing is based on matching template rules—defined by `template` elements—against the input XML tree. Each `apply-templates` instruction selects a set of nodes in the input tree, and matches the template rules against each of the selected nodes. Every node must—after precedence resolution—be matched by exactly one template rule, and this rule is then instantiated with the given node as *context node*. The produced result fragments for each of the selected nodes are concatenated according to the document order of the input nodes and inserted at the place of the `apply-templates` instruction. *Built-in template rules* ensure that a selected node is guaranteed to always match some rule, although the result of instantiating a built-in rule is often unwanted as all string values in text and attribute nodes are copied directly over. It may occur that more than one rule matches a selected node, and such an error should be corrected through *priorities*, which is simply a number specifying precedence between the template rules. The template instantiation process is continued recursively until no more XSLT instructions need processing.

The template instantiation process is initiated by matching the template rules up against the root node. The final result document will be the result of instantiating this initial root-matching template rule.

The elements of XSLT go in three categories: Document elements, top-level elements and instructions. Either of the document elements `stylesheet` or `transform` can be used to define the transformation. The document element contains all the top-level elements, and defines some global properties

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

  <xsl:template match="news">
    <html>
      <head>
        <title>News</title>
        <meta http-equiv="Pragma" content="no-cache"/>
      </head>
      <body>
        <xsl:apply-templates select="item"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="item">
    <xsl:apply-templates select="headline"/>

    <font face="arial,helvetica" color="#808080" size="1">
      Den
      <xsl:apply-templates select="@date"/>
      Kl.
      <xsl:apply-templates select="@time"/>
    </font>
    <br/>

    <xsl:apply-templates select="text"/>

    <hr/>
  </xsl:template>

  <xsl:template match="headline">
    <font face="arial,helvetica" size="4">
      <b>
        <xsl:value-of select="text()"/>
      </b>
    </font>
    <br/>
  </xsl:template>

  <xsl:template match="text">
    <b><xsl:value-of select="../@category"/></b> -
    <xsl:apply-templates select="p"/>
  </xsl:template>

  <xsl:template match="p[1]">
    <xsl:value-of select="text()"/>
  </xsl:template>

  <xsl:template match="p">
    <p>
    <xsl:value-of select="text()"/>
    </p>
  </xsl:template>
</xsl:stylesheet>
```

Figure 2.4: An example XSLT transformation, presenting news headlines in XHTML format.

such as the version of XSLT being used. The top-level elements are all the document-wide constructs such as template rules, output format specifications and so on. They may only appear as children of the document element. Finally, the instructions are all those elements occurring in XML templates, which construct the output. Extraordinarily the `variable` and `param` elements can occur both at top-level and in templates - inferring different meanings according to their place - and are therefore both top-level elements and instructions.

XSLT makes heavy use of the XPath language, and it is XPath that is used to select nodes for template rule matching. It has been slightly expanded by including the *result tree fragment* data type. Values of this type basically—as the name implies—describe a fragment of the result tree, in contrast to the node-sets which refer to the input tree. These result tree fragments can be stored in variables and re-used throughout the transformation, but they can not be manipulated. They can only be copied directly into another result fragment, or be coerced to one of the basic data types.

XSLT has no small number of features, the most important being:

- The template rules can be partitioned into named disjunct sets called *modes*. Each `apply-templates` instruction must address one of the modes exclusively. The modes are commonly used to form a sort of state machine, where the processing depends on the which state it is in.

- XSLT has a rather elaborate *inclusion* and *import* mechanism. Included documents are treated as if actually written in the including document. Imports however are subject to *import precedence*, which basically states that constructs from imported documents have lower priority than non-imported ones, and the last import takes precedense over the first import.

- Variables and parameters allow re-use of values, as well as passing values around in the document. Top-level parameters are assigned a value by the XSLT processor at instantiation. While the type of variables can be determined statically from their definitions, parameters are weakly typed in the sense that they can recieve any type of value. An elaborate system for value coercion enables the processor to always coerce values to the required type.

- A number of well-known flow-control constructs from imperative languages have their place in XSLT: `if`, `for-each`, `choose` and `call-template`. The `call-template` instruction is reminiscent of a procedure call, as it is used to call a specific template rule, typically with parameters. The rest of the instructions function much like their

imperative counterparts, except that the code blocks have been replaced with XML templates to be instantiated.

- The `element`, `attribute`, and `value-of` instructions are for constructing output nodes dynamically. `value-of` iunstructions are typically used for copying string values over from the input. `element`, `attribute` instructions can—in conjunction with *attribute value templates*—be used to construct elements and attributes with computed names.
  Attribute value templates generally just allow one to perform dynamic attribute value construction. For example, the attribute definition: `date="{@day}.{@month}.{@year}"` constructs the value of a date attribute from a number of attribute values in the input document. On literal result elements, this is simply a sugared form of an `attribute` instruction. When used in attributes of other XSLT constructs, it is a different matter.

The template rules and `apply-templates` instructions of XSLT form a quite simple core of XSLT resembling *structural recursion on trees*[11]. However, the full language has many features, and there are many corners of the language that one can stumble upon. To name an one, the `for-each` instruction seem at first sight simply to be an embedded template rule. However, the `for-each` instruction is in fact able to recurse on namespace nodes, which normal template rules is not. The `for-each` does not have to go through the filtering of a template rule match expression.

For more details on the language, we refer to the specification [23].

2.8 Summary

We have now been introduced to the XML Architecture and its many standards. The core standards have been described in detail, and the important subject of XML classes and schema languages has been examined.

The XPath data model has been identified as a suitable reference model for describing XML data abstractly, and it will be the basis of discussion throughout the rest of the thesis.

Finally the XML transformation language XSLT, and its related standard XPath, has been looked into. Like with any other language of this complexity, errors will creep in, and the question whether we can find these errors statically comes to mind. This question will be the topic of the following chapter.

[11]Structural recursion: The process of recursively applying some transformation rule on the nodes of a tree, starting from the top.

3

Static Output Validation of XSLT

This chapter examines the problem of static output validation of XSLT, how a solution might be approached, whether the problem is already solved, and to what extent it is solvable at all.

3.1 Motivation

The many XML applications which one can find on the Web all define small languages, essentially sets of XML documents, which describe data of some sort. These XML languages are often formalized with schemas, but even if they are not, some specific format is still expected of documents in the language.

XML languages are used in many ways. Often stylesheets are provided for presenting the data in some way, for example as HTML. Or the XML data can be used as intermediate values in application *pipelines*[1]. The latest trend seems to be XML *Web APIs*[2], such as the "Google Web API" [38] and "Amazon Web Services" [4].

Unfortunately, programming errors are inevitable, and the XML applications can fail in many ways. One of these ways is *recieving invalid input*. If an application for example queries the Google Web API for a search, it expects the data to be of a certain format, as specified by Google. If the query result is not of this format, the application will most likely fail either *explicitly* with

[1]Pipeline: A sequence of applications each using the the results of the previous application in the line.

[2]Web API: Application Programming Interface over the internet

an error message, or *implicitly* through arbitrary behavior. Especially with the application-to-application Web services the problem becomes apparent, as also the output of the Web service programs almost certainly will lead to failures. And perhaps worse: the recieving applications can do nothing about the error, except contact the Web service authors.

The practice of today regarding debugging of XML programs seems mostly to be test-runs of the program, possibly combined with runtime schema checks to at least make the errors explicit.

At this point however one can only speculate whether these checks could be done *statically* instead. That is, examine, simply by looking at the program, whether the documents always conform to some schema. Given two programs where one, the *sender*, produces some XML document used as input in the other program, the *reciever*, we wish to statically verify that the reciever can never fail because of invalid input from the sender. For this, we must analyze the sending program.

Statically validating the conformance of XML output for some program would offer us two things: (1) Fewer runtime errors, and (2) possibly skipped runtime schema conformance checks, in the recieving programs. More precisely:

- the explicit or arbitrary errors resulting from invalid input in the recieving programs do not occur, and
- if runtime conformance checks are performed, and if the same or a stronger schema is used in the static validation as in the runtime conformance checks, the runtime checks will be superfluous, and can be removed.

Regarding the case of static analysis with a weaker schema: Statically validating of the output may be too difficult with the more expressive of the schema languages, so the static analysis might only be able to validate against a lesser schema. In this case, the amount of runtime errors is certainly reduced, but the runtime checks are not superfluous as they otherwise would be.

Among the many XML manipulation languages, XSLT is certainly one of the most popular. Perhaps because it was the first such language by the W3C, and perhaps because it is molded about presenting XML data as XHTML, which is an often needed tool when working with XML.

There is lots of static validation work on various XML manipulation languages out there, which we will examine in more detail in Section 3.3, and some nice results have been achieved. In particular, XSLT has recieved quite some attention towards static analysis of output and other types of analyses,

but as we shall see, most of the results tend to work with less useful fragments of XSLT. In any case, there has yet to be developed a practical tool capable of statically validating the output of XSLT transformations written in practice. We intend to remedy this.

3.2 The Problem

Having identified this unresolved issue for XSLT transformations, we wish to create a tool that can statically validate conformance of the output of XSLT transformations to some schema, or in case the transformation output is not always valid, present the user with comprehensible error reports.
But let us first define what we mean by an *XML transformation*:

> *An* XML transformation *is a program that takes one or more XML documents* $X_{in}^1 \cdots X_{in}^n$ *as input and, if it terminates, produces one or more new XML documents* $X_{out}^1 \cdots X_{out}^m$.

Note that although we require a transformation to take at least one input document, we do not require that it is used. We will refer to XML transformations as simply transformations, when there is no ambiguity.

Though a number of properties might be of interest with regard to static analysis of transformations, we have chosen to examine the apparently troublesome problem of determining conformance of the output document to some schema. Let us define the property in question as *output validity* with respect to some input and output schemas:

> *Given input schemas* $S_{in}^1 \cdots S_{in}^n$ *and output schemas* $S_{out}^1 \cdots S_{out}^m$, *an XML transformation* T *is said to be* output valid with respect to $\mathrm{S}_{\mathrm{in}}^1 \cdots \mathrm{S}_{\mathrm{in}}^{\mathrm{n}}$ and $\mathrm{S}_{\mathrm{out}}^1 \cdots \mathrm{S}_{\mathrm{out}}^{\mathrm{m}}$ *if and only if all sets of output documents producable by a terminating execution of* T *conform respectively to* $S_{out}^1 \cdots S_{out}^m$ *given that the input documents conform respectively to* $S_{in}^1 \cdots S_{in}^n$.

The concept of output validity is often referred to as *type checking*, but that particular wording can be misunderstood if the transformation language works with types in some other way, and many transformation languages do.

The problem we shall be tackling in this thesis is output validity with respect to only a single input document and producing only a single output document. In this case, the problem of deciding output validity with respect to schemas S_{in} and S_{out} can be formulated algebraically as: *Given a transformation* T*, input and output schemas* S_{in} *and* S_{out}*,* T *is output valid if and only if* $T(\mathcal{L}(S_{in})) \subseteq \mathcal{L}(S_{out})$. In this definition, we view the transformation T

as a function mapping XML documents to XML documents. The *language*, $\mathcal{L}$, of some schema is the set of XML documents conforming to that schema.

Statically determining the truthfulness of this property for some transformation may not be so easy however, and often analyses must make use of *approximations* in order to at least give *some* results rather than none. A *conservative approximation* of the output validation problem will roughly speaking give an answer in either {INVALID, MAYBE VALID} or in {MAYBE INVALID, VALID}. The difference is that the former is able to guarantee *invalidity*, while the latter is able to guarantee *validity*. Furthermore, the former reports only definite errors, while the latter may produce spurious error reports. But there shall be no doubt that determining the truth of the property—i.e. guaranteeing output validity—is by far the most useful result in this context. Every transformation is supposed to be output valid. Recieving the guarantee allows one to lay it to rest.

In contrast to the approximations, an analysis that is able to determine *exactly* when a transformation is output valid or not, is said to be *sound and complete*.

We wish to analyze output validation for XSLT transformations. This will be on the current version 1.0 of XSLT. We do not claim to be able to make a complete analysis, but in order to still be able to make guarantees about the output of transformations, we must approximate conservatively such that we answer "valid" only when the transformation is in fact output valid.

We may not be able to issue such guarantess for the more complex transformations. In such cases, the transformation in question can sometimes be simplified in order for the analysis to better understand the transformation, and thus perhaps be able to produce the guarantees. The more precise the analysis, the less of a problem this will be.

It would be unrealistic to tackle this problem for all the schema languages out there. A single model has to be adopted. Examining the problem at hand, it stands out that the more expressive the output schema model, the harder it will be to solve the problem but the more potentially useful the result. The usefulness potential stems from the fact that a more expressive model is able to cover more classes of XML documents accurately. Regarding the input schema model, a more precise input description has the potential to produce more accurate analysis results, since we are given more information for narrowing down the possible outputs.

But it is not given that this problem can be solved satisfactorily at all for a more expressive ouput schema model. So it seems sensible to start at the bottom and work our way up.

Furthermore, this project being of a pragmatic nature suggests the use of not just a formal model, but rather a schema language in practical use. Un-

fortunately no consensus has been achieved on a common schema language, except perhaps DTDs. DTDs are inherently weak and lack important features. But it is part of the XML standard itself and is a derivative of the DTDs of SGML, which has been around for quite some time. In effect, DTDs are widely used either as the main schema language where possible, or as an easily human readable and well-understood alternative schema. Such alternative DTDs will most often express a superset of the language described by the main schema, so relying on them for analysis, rather than the main schema, loses some precision.

Given these considerations, DTD has been chosen as the schema language of this project in spite of its weaknesses. Essentially, DTD as output schema model shall ensure the tractability of the problem. We have already chosen the transformation language to be analysed, but a less expressive schema model can reduce the complexity of the problem as well. DTDs for input are at this point a natural choice for uniformity. Their simplicity will also make extracting information about the input easier. It will remain for future work to try and extend the analysis to more expressive schema models, as well as to start handling namespaces. Namespaces are not supported by DTDs, but are an important part of the XML architecture today.

The availability of schemas for the various transformations found on the Web shall be important for experimenting with practical examples. If schemas are missing, DTDs can be produced through XML-to-DTD generators such as the SAXON DTDGenerator [49]. Schema-writing is often kick-started this way by automatically generating an initial schema from sample XML files. However, such tests are of secondary quality. While they do help examine the precision of the analysis on practical tranformations, they do not help identifying what output validity errors occur in practice. The intended types can not be known simply from example XML files.

Finally, we wish this work to be a help for XSLT development in practice, so:

> *The analysis should be as correct and quick as is necessary for it to be practically useful in day-to-day development of XSLT transformations.*

This means that we must test our work on a decent number of XSLT transformations randomly picked on the Web. It also seems appropriate to require the transformation to be able to handle the *identity* transformation. Realising that such an identity can be written in many ways in XSLT, we define the *general identity* as the most generally written identity that copies all input to the output exactly, and which is completely independent of the class of XML documents it recieves as input. Such a tranformation can be seen in Figure 3.1. Another variation of the general identity can be found in the

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

<xsl:template match="/|child::node()|attribute::node()">
  <xsl:copy>
    <xsl:apply-templates select="child::node()|attribute::node()"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Figure 3.1: The *general identity* transformation. Note that namespace nodes are copied automatically with the elements.

XSLT specification.

The work of Audebaud and Rose in "Stylesheet Validation" [5] indicates an interesting definition of the output validation problem. In this definition we allow specifying more than one schema for each input or output document. I.e. the documents in question may conform to any one of a set of schemas. Or in yet other words: The document must conform to the union of those schemas.

Adopting a schema model that is closed under union means that this is no issue, since the sets of schemas can be merged into one. Unfortunately, schema languages in practice often do not enjoy this property of closedness under union. This includes the schema languages DTD[3], DSD2[4], XML Schema. *Regular tree automata* [27] on the other hand are closed under union [27]—and thus also the *regular expression types* of XDuce [41] and *specialized DTDs* [64]. So is Relax NG [26].

However, the practical relevance of this idea of multiple schemas for a document is doubtful, as transformations seldom are intended for more than one kind of input or to produce more than one kind of output. An example is the *general identity* transformation (Figure 3.1), which is rather artificial, and for which such multiple-schema analysis is somewhat pointless: It can be analyzed one schema at a time.

Not dismissing the notion entirely, it is clear that a multiple-input-schema transformation is trivially validated by a single-input-schema validator by simply running the validation for each input schema. If each run guarantees valid output, then the output is valid for the union of the schemas. Multiple output schemas on the other hand are not as easily handled, unless of course the schema model is closed under union. It can not be done simply by calling a single-output-schema validator as a black box.

[3]Given a DTD with the two content models `a: (b)` and `b: (c)`, we can not express a union of this, with the content models `q: (b)` and `b: (d)`, in the DTD language. The content model of `b` would depend on its parent. Note however, that *specialized DTDs* [64] describe precisely the extension needed for DTDs to express the above union.

[4]DSD2 can not express a set of legal root elements. Only one or any.

Since DTDs are not closed under union, we can not trivially handle multiple schemas for the output, but as these multiple schema transformations have very limited or no practical use, we ignore the issue. We allow specifying only one input and one output schema.

We now conclude with the PROBLEM STATEMENT of the thesis:

> *Given DTDs D_{in} and D_{out}, respectively describing the expected input and output of a single-input XSLT 1.0 transformation T, we wish to approximate conservatively whether T is output valid with respect to D_{in} and D_{out}. The analysis shall answer "valid" only if T is guaranteed to be output valid, and otherwise report error messages that may or may not be actual inconsistencies with the output validity of T. The analysis must further be practically useful in day-to-day XSLT development.*

Note that XSLT 1.0 transformations only are able produce a single output document. This is not the case with XSLT 2.0 [48], which is currently in a working draft.

3.3 Related Work

As discussed in Section 3.1, testing the output validity of programs often amounts to running the application on various input, and examining the output.

A more systematic approach are the step-by-step debuggers. A number of these debuggers for XSLT exist today, including: XSLDebugger [72], Xselerator [55], XSLT-process [68], XSLTracer [47], CodeX [7], xsldbg [43], XML Spy [3]. They mainly consist of an XSLT processor able to run transformations in small steps, along with some visual tool to inspect the state of the processor along the way.

These attempts are clearly inspired by the traditional approach on debugging programs in general-purpose languages, but XML manipulation has some special characteristics separating it from the general programming: The exact composition of input as well as inteded output is likely known, paving the way for our type checking ideas. And the above approaches seem pitiful compared to the idea of statically *guaranteeing* valid output simply by looking at the transformation and the input and output schemas.

One early optimistic attempt on static output validation of XSLT while still in a *working draft* state was "Stylesheet Validation" [5] by Audebaud and Rose. The analysis is given input and output *DTD patterns*, an extension of DTDs essentially describing sets of DTDs. Through a set of *typing rules* they try to establish guarantees about output validity for an XSLT transformation.

While their goals are ambitious, the method is unfortunately only applied to a small toy-example, effectively reducing their work to the tiny fragment of XSLT that this example represents.

Although the attempt had little success, it perhaps helped the community to define the problem and realize the difficulty of it. And as mentioned in Section 3.2, they did have an interesting angle on the output validation problem.

The problem is certainly not trivial. As Kepser proved in "A Proof of the Turing-completeness of XSLT and XQuery" [50], XSLT is Turing-complete, making Rice's theorem [71] apply. Thus, any interesting question about the behavior of an XSLT transformation is undecidable.

But that does not mean we can not get some good results by approximating. Kepser's proof basically relies on template calling, parameter passing, and some simple arithmetic and string handling, so we can expect analyzing these features to be hard. But we note that those features seldomly are used to shape the processing of documents. That job usually lies with the template rules, `apply-templates` instructions, and other basic constructs.

An interesting line of research has carried the static output validation problem into the field of tree automata and tree transducers (see [27] and [62] for a further treatment of the subject). The basic idea is that XML data can be viewed as ordered and labeled trees, and thus the established research on tree automata and transducers can be applied to XML as well. This twist of events has to some degree revived the research on tree transducers.

The beauty in this approach lies in the the elegance and simplicity of the tree formalisms. It—in a way—examines the essence of the problem. It exposes the essential features of XML schema validation (automata) and transformations (transducers).

In light of the many and volatile XML transformation languages in the field, Milo et. al. propose in [58] a tree transducer formalism called *k-pebble transducers*, attempting to cover all the languages in one model. This model is able to simulate most XML transformation languages (here called *query languages*) when ignoring text and attribute values (collectively called *data-values*). This expresses the core of languages such as XSLT, XML-QL [31], and the like.

Schemas are modeled through *regular tree expressions*, which are equivalent to *regular tree automata*, and which therefore are equivalent in expressive power to *specialized DTDs* [64]. This of course means that they subsume DTDs.

Sound and complete static output validation is then performed on these transducers through *inverse type inference*. That is: Given a transducer T and an output type $\mathcal{S}_{out}$, find the type ${\mathcal{S}_{out}}^{-1} = T^{-1}(\mathcal{S}_{out})$. Then check whether $\mathcal{L}(\mathcal{S}_{in}) \subseteq \mathcal{L}({\mathcal{S}_{out}}^{-1})$. This novel approach proves to make the static output validation problem tractable for k-pebble transducers, and it has

since been adopted in a number of works, as we shall see.

The static output validation problem is proven decidable for the k-pebble transducers, but with *non-elementary* time complexity[5]. This result can hardly be called of practical interest, without empirical study showing otherwise, but the work has definitely helped us understand the nature of the problem.

Later work in "XML with Data Values: Typechecking Revisited" [2] further examines decidability for static output validation of XML transformation languages when text and attribute values are present. The boundaries of decidability are traced by trying a number of more or less expressive formalisms for schemas and transformations. They mostly center around DTDs and XML-QL like transformations.

It turns out that static output validation quickly turns undecidable when comparisons of text and attribute values are allowed, and the authors are forced to restrict their schema and transformation models tightly in order to keep the decidability. The results again yield results of very high time complexity. One of the best results in this direction are *PSPACE* complexity when restricting to non-recursive transformations and non-cyclic input DTDs[6].

In "Towards static type checking for XSLT" [79], Tozawa examines a fragment, XSLT0, of XSLT, covering the structural recursion core of XSLT. Inspired by the work of Milo et. al. in [58], exact static output validation is performed through inverse type inference. The problem is found to be decidable in exponential time complexity, and even claimed to be fast in practice.

However, the fragment only contains simple child steps in the recursion, and attributes are disregarded. XSLT0 is unsuited for a reduction of full XSLT transformations to XSLT0. This can be achieved for only very basic transformations. And finally, Tozawa presents insufficient experiments to back the claim of practically useful performance.

Where Milo et. al. [58] examined decidability for the exact static output validation problem, Martens and Neven investigates in "Typechecking Top-Down Uniform Unranked Tree Transducers" [57], the polynomial time computability of the problem. More precisely, it is examined to what degree tree transducers and their types must be restricted in order to achieve PTIME complexity. PTIME is finally achieved with DTDs by restricting *copying power* and *deletion* for top-down uniform unranked tree transduc-

[5]Non-elementary time: Not bounded by a fixed number of compositions of exponentials.

[6]Non-cyclic DTDs: No element can have itself or an ancestor as its child. Such cycles result in trees of arbitrary depth. XHTML for example is cyclic.

ers: The number of recursions in transformation rules are bounded, and every input node must produce an output node. The transformer moves strictly downwards and it cannot distinguish the order of siblings.

The results seem only of theoretical interest considering the heavy restrictions imposed on the transducers.

Looking back, it would seem that trying to perform sound and complete static output validation on even very simple models of tree transformation is of too high a computational complexity to be feasible. We have however seen some results with low enough complexity so as to indicate such practical usefulness, but the indications have been backed by very limited empirical study. And even if they turn out to be practical, the computational models have such a poor expressiveness that it will make it very impractical to program directly to these models. Also trying to approximate transformations in contemporary languages, such as XSLT, into the models will prove difficult, as the models lack vital features. In the particular case of XSLT, the restriction to top-down traversal seems to be difficult, and the restrictions on deletion we have seen seem out of the question. Work will definitely have to be done if such approximations shall be proven useful.

However, we do note that practicality and especially approximation have not been the primary focus in most of these papers, and the theoretical aspects are certainly of interest.

On the more pragmatic side, we shall examine a number of XML manipulation languages for which various static analyses, including static output validation, has been an important design goal. These languages can serve as an inspirational source for our analysis of XSLT.

In XDuce [41], Hosoya describes a functional and statically typed XML manipulation language. The static typing ensures correct use of XML values, as well as output validity. The type system is based on *regular expression types*, which are equivalent in expressiveness to regular tree automata. This results in decidable and exact algorithms, but also in exponential time complexity on the subtyping checks. However, Hosoya seems to have achieved acceptable performance in practice by applying a number of heuristics [42]. Furthermore, the XDuce type system has been extended with constraints on attributes in [40].

The work on XDuce has inspired other work in the field. Most closely related is ℂduce [8], which extends the type system of XDuce with higher order functions as well as intersection, and complent type operators. The Xtatic project [67] attempts to integrate the type system of XDuce, into the C# general purpose programming language.

In "A Type-safe Macro System for XML" [66], Perst and Seidl design a language for describing small XML macros, meant to alleviate users from the quite verbose syntax of XML. The macros closely resemble named tem-

plate rules of XSLT with template parameters of type result tree fragment, and with only static recursion through `call-templates` and fragment insertion through `copy-of`. The macro language additionally has a construct for passing strings for attribute values. Although Perst and Seidl prove that the language is not expressible with the k-pebble transducers of Milo et. al. [58], the language is quite limited: Recursion and mutual recursion between macros is not possible, and neither is deconstruction or any other kind of probing of the input. This leaves only rigid insertion of whole argument fragments in an input independent manner. The exact output tree of a macro call can be statically determined.

Various typing issues is examined for the macros. The static output validation problem is shown decidable in exponential time through the Milo et. al. inspired inverse type inference problem. Further, a faster type inference algorithm is developed for the macros. However, attributes are left entirely out of the static validation, and little experimentation is performed to support the claim of practical usefulness.

The `<bigwig>` project [14] takes an interesting approach on Web service programming by developing a language with direct high-level support for *sessions*, which is a series of connected Web pages, with for example forms for interacting with the service. Furthermore, the Web pages are in `<bigwig>` constructed dynamically from HTML fragments and computed values. A powerful static analysis ensures correct use of HTML values, and guarantees valid HTML output.

In the following JWIG project [21][20], the work of `<bigwig>` is taken to a more pragmatic setting, by infusing the high-level Web page construction—this time in XHTML—into the highly popular programming language Java. In both the `<bigwig>` and JWIG projects, static analysis of HTML/XHTML values are performed through the unique concept of *summary graphs*, first defined in “Static Validation of Dynamically Generated HTML” [13]. These graphs describe how the HTML/XHTML fragments are inserted into each other, as well as some specific knowledge about the use of the programming constructs in the analyzed programming language. Through control and data flow analysis, a summary graph is constructed, conservatively approximating possible output of a program. This allows for quite exact and efficient static output validation of `<bigwig>` and JWIG programs.

The Xact [51] framework further extends of the work in JWIG to development of application-to-application Web services. In this setting, not only construction of general XML values is needed, but also *deconstruction* of especially XML provided as input to the service. Xact provides an extension to Java with such high-level manipulation of XML values, and—like its predecessors—provides static output validation, as well as analysis of the use of intermediate XML values. The analysis uses a summary graph model modified to suit the new setting, still using the summary graph analysis of

JWIG. Of particular interest is the Xact's ability to analyze deconstruction of not only the input XML values, but also intermediate constructed values. Also note that analysis of XML values in Xact are performed on arbitrary XML, with DTDs as input and output types.

A second XML manipulation language out of the W3C is XQuery [9][33]. It is basically SQL in an XML setting, and takes more of an information retrieval angle on the issue, compared to XSLT. Recognizing the usefulness of static guarantees, XQuery defines, as part of the standard, a set of type inference rules, allowing conservatively approximated static output validity guarantees to be made. This is particularly interesting, since XQuery is one of the first languages from a standards organization, using such formal methods. XQuery is still in a working draft state.

The next generation XSLT language, XSLT 2.0 [48], introduces some static type checking as well. By supplying the transformation with type declarations, the XSLT 2.0 processor may detect type errors statically. This functionality is optional in the XSLT processors, and the static type checking procedure is left entirely to the implementors. The language is still in a working draft state, but unless the specification becomes more specific, the static type errors will likely never move beyond trivial literal element validations in the template rules.

In "Static Analysis of XSLT Programs" [32], Dong and Bailey pursue a different aspect of debugging XSLT transformations. Given an XSLT transformation and an input DTD, they perform static analysis of the flow of template instantiations for the transformation, in order to examine various properties such as *unreachable templates*, *termination*, and others. Their analysis is conservatively approximating, so that the errors reported are guaranteed to be actual errors in the transformation. This means, for example, reporting template rules which are guaranteed to never be instantiated.

Rounding up, we can see that there are a number of XML manipulation languages successfully producing static output validity guarantees, which seem to perform well enough for practical use. The trouble is that it is very hard to gain acceptance with such languages, even though your work is practically useful. By attacking the problem for an already popular language, we do not have this problem. But at the same time, it leaves us incapable—perhaps to the better—of forming the language to our analysis. We have to form our analysis around the language, and this can prove troublesome.

We shall try and apply some of the best static analysis ideas in the field of XML manipulation languages, in order to achieve a practically useful static output validation algorithm for XSLT.

3.4 Goals and Restrictions

In recapitulation, let us state our goals for the output validation analysis of XSLT tranformations in a concise form:

- Generally the analysis should be as precise and fast as necessary for it to be practically useful in day-to-day development of XSLT transformations.
- It should be able to verify output validity of the *general identity* transformation depicted in Figure 3.1.
- The validator should be able to verify validity of, or identify errors in most practical XSLT transformations.

On the flipside we have established some restrictions to make the project doable:

- We only handle XSLT 1.0 transformations with a single input document.
- Only one input and one output schema can be specified.
- Input and output schemas must be DTDs.
- We do not analyze the text output method, since there is simply no schema to check against in this case.
- The transformation must use only one input and one output namespace. Since DTDs do not support namespaces, we can only analyze single-namespace output transformations.
- We do not handle `disable-output-escaping`. Analyzing this would require complex string analyses, and then trying to figure out what XML it can represent. This would be a project in itself. And quoting the XML specification: "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."
- We do not support extension elements or functions except the ones we introduce ourselves.
- We do not support the `namespace` axis. The axis is troublesome in certain situations, as briefly discussed in Section 2.7. Furthermore, selecting namespace nodes is very seldom used, and it seems pointless given that the DTDs are unaware of namespaces.

3.5 Summary

The concept of output validity of XSLT transformations has been defined as having the output of the transformation conform to some XML schema. Such conformance is often expected of the output, but is hard to verify statically. The usefulness of statically verifying output validity has been discussed, and related work in the field has been examined. No practically useful analysis technique has emerged for XSLT, but some interesting ideas have been applied to related languages which might be of use. Finally some reasonable expectations of a static output validator has been set, to establish a foundation for design of the analysis.

4

Analysis and Design

With the problem to be solved and some goals and restrictions in hand, we shall now design a static output validation analysis, while attempting to meet our goals.

4.1 The Analysis Approach

The basic idea behind our approach to the static output validation problem for XSLT stems from the striking similarity between the XSLT processing model and the output document construction in the `<bigwig>` [14], JWIG [21], and Xact [51] languages. In Xact—which is the most recent and most closely related of the three languages—you construct XML output basically by *plugging* well-formed XML fragments into each other. Insertion points are defined by named *gaps* in the fragments. The construction process starts out with a number of constant XML fragments and possibly some fragments extracted from input documents, and then simply proceeds by plugging these XML fragments into one another until the desired result XML document has been produced. Furthermore, strings can be plugged in as attribute values or as plain text.

Similarly the XSLT processing model consists, as described in Section 2.7, of a number of constant fragments as well as one or more input documents. The fragments contain XSLT instructions which upon execution will be replaced by some other XML fragment or plain text. So the key here seems to be insertion of XML fragments and strings into designated points in XML fragments. In Xact, the insertion process is described by a Java program. In

XSLT, it is more or less described through interconnecting `apply-templates` selections and subsequent template rule instantiations. The Xact static analysis tries to figure out the flow of the XML fragments in the program through classical control and data flow analyses. The result of these flow analyses are a number of *summary graphs*, expressing a conservative approximation of each intermediate XML value, and of the possible output for each exit point[1] in the program. The approximation is conservative in the sense that the summary graph is guaranteed at least to express all actually possible output. This sounds much like what we want of the XSLT analysis.

The summary graph concept shall be of direct use to us in analyzing XSLT. In order to construct such a graph representing a conservative approximation of the possible output of an XSLT transformation, we must be able to figure out the *flow of construction* in the transformation. XSLT is not an imperative language like Java is, although it does have some constructs with similar semantics. We shall instead perform a more ad hoc analysis of the flow in a transformation by connecting each `apply-templates` instruction with the possible matching template rules. This should cover the core of the language, but the many other language features will also need to be handled. Some, such as the `call-template` instruction referring to a named template rule should be easy. Others, such as the `element` instruction has the potential to be very hard to analyze.

With the flow of template instantiations exposed, we shall try and construct the final summary graph as accurately as possible, by modelling the order and cardinality of the XSLT template instantiations. This final summary graph will be referred to as the *output summary graph*. Let $\mathcal{L}(SG)$ refer to the set of XML documents which the summary graph SG represents. Given an XSLT transformation T and its respective input and output DTDs D_{in} and D_{out}, it must hold for the output summary graph SG that: $\forall x \in \mathcal{L}(D_{in}) : T(x) \in \mathcal{L}(SG)$. Determining output validity is then a matter of checking whether $\mathcal{L}(SG) \subseteq \mathcal{L}(D_{out})$. The better we are at figuring out the flow and construct the output summary graph, the better the chance that that last property will hold. If it does not hold, we shall print error reports of what might be wrong with the given transformation. These reports may be true errors, or they may be spurious. We must strive to create the output summary graph as tightly around the actually possible output as possible, so as to generate as few spurious error reports as possible.

Assumptions

The first layer of errors which can be expected in XSLT is simple mistakes leading to invalid XSLT, in the sense that the written transformation does

[1] Here we refer to the *show* and *exit* statements which produce the output documents.

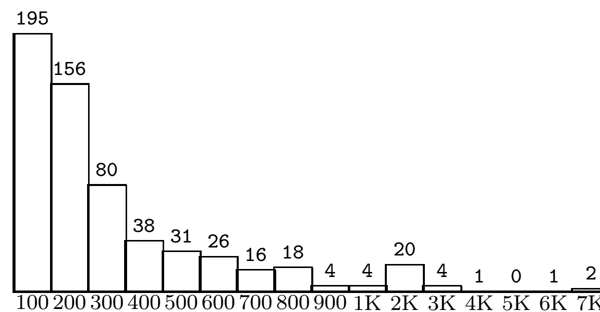


Figure 4.1: XSLT transformation sizes in number of lines.

not conform to the XSLT specification. These errors are generally easy to find, and most XSLT processors will do some form of conformance checking during parsing of the transformation document. The focus of this project is on the more subtle errors, and we shall, for the sake of keeping to the point of the problem, assume that trivial errors such as lacking specification conformance have been taken care of. More precisely:

- We assume the XSLT transformation conforms fully to the XSLT 1.0 [23] and XPath 1.0 [25] specifications, as well as any other specifications referred to by these two. Note that this says nothing about the schema conformance of input or output documents to the transformation. The XSLT specification is unconcerned with schemas except describing the XSLT language itself.
- We assume the input and output DTD to conform fully to the XML [18] core specification, as well as any other specifications referred to by it.

These assumptions are a natural extension to the limitations established in Section 3.4. Note that specification conformance not only establishes the basic syntax, but also includes errors such as referencing variables that have not been defined.

4.2 Data Mining

In order to keep the pragmatic focus of the analysis, let us begin with some data mining on how XSLT is used in practice. A rather large amount of XSLT transformations have been gathered from the Web to represent XSLT use in practice. This may only be a tiny fraction of what has been written, as many transformations are not shared with the public, but it should suffice for giving us a general idea of the use.

A total of 596 stylesheets have been gathered, amounting to 185,117 lines of XSLT code written by hundreds of different authors. The sizes of the stylesheets distribute as illustrated in Figure 4.1. As can be seen, the

Top-level Element	Number	Fraction
template	8,769	73.344%
attribute-set	1,034	8.648%
param	793	6.633%
variable	576	4.818%
output	378	3.162%
include	180	1.506%
import	116	0.970%
strip-space	57	0.477%
key	20	0.167%
constant	9	0.075%
preserve-space	8	0.067%
function	6	0.050%
namespace-alias	4	0.033%
if	2	0.017%
decimal-format	2	0.017%
script	2	0.017%

Instruction	Number	Fraction
value-of	12,333	16.940%
text	9,953	13.671%
apply-templates	8761	12.033%
when	8,653	11.885%
if	5,921	8.133%
attribute	5,853	8.039%
call-template	4,228	5.807%
with-param	3,911	5.372%
choose	2,913	4.001%
otherwise	2,535	3.482%
variable	2,070	2.843%
for-each	1,878	2.579%
param	1,455	1.998%
element	938	1.288%
copy-of	356	0.489%
sort	229	0.315%
number	221	0.304%
message	173	0.238%
comment	161	0.221%
copy	123	0.169%
processing-instruction	28	0.038%
apply-imports	26	0.036%
use	25	0.034%
eval	24	0.033%
result-document	10	0.014%
document	8	0.011%
template	7	0.010%
sequence	6	0.008%
entity-ref	2	0.003%
fallback	2	0.003%
pi	1	0.001%
script	1	0.001%
for-each-group	1	0.001%

Figure 4.2: The distribution constructs used in our gathered XSLT transformations.

sizes of transformations are typically below 800 lines, which means we should aim for being able to handle at least such sizes.

In Figure 4.2 the distribution of top-level elements and template instructions can be seen. Naturally, most of the top-level elements are template rules, but apart from that there seems to be an unfortunate tendency for using top-level parameters. If these are used to recurse on, or if they are copied to output, we can say nothing about the results. We can only hope, for the sake of statically analyzing XSLT, that most of these parameters are used for simple string values or something similar.

On the instruction side, we can see that `apply-templates` instructions take care of most recursion, but perhaps not as much as could be expected of the main recursion construct. Developers certainly have a tendency to transfer their imperative programming habits into XSLT. Fortunately `sort` instructions, which express complex ordering which would be very hard to analyze, are seldom used. `copy` and `copy-of` are equally seldom used, indicating that the input of the transformation rarely is of the same type as the output. Also, a testament to bad programming habits, a number of constructs and instructions, which are not part of the XSLT 1.0 standard, litter the XSLT tranformations.

We have further examined the use of select expressions on the core XSLT

Select Category	Number	Fraction
default	3,393	30.9%
`a`	3,326	30.3%
`a/b/c`	1,163	10.6%
`*`	758	6.9%
`a	b	c`	474	4.3%
`text()`	235	2.1%
`a[...]`	223	2.0%
`/a/b/c`	115	1.0%
`a[...]/b[...]/c[...]`	82	0.7%
`@a`	68	0.6%
`/a[...]/b[...]/c[...]`	43	0.4%
`..`	32	0.3%
`/`	8	0.1%
`$a`	371	3.4%
name known	254	2.3%
parent and name known	200	1.8%
set of names known	73	0.7%
sibling known	31	0.3%
set of parents known	12	0.1%
parent known	9	0.1%
nasty	124	1.1%

Match Category	Number	Fraction
`a`	4,712	53.7%
absent	1,376	15.7%
`a/b`	523	6.0%
`a[@b='...']`	455	5.2%
`a/b/c`	446	5.0%
`/`	254	2.9%
`*`	218	2.5%
`a	b	c`	179	2.0%
`text()`	52	0.6%
`@a`	24	0.3%
`@*`	16	0.2%
`a:*`	13	0.1%
`processing-instruction()`	11	0.1%
`@n:*`	4	0.0%
`a[...]`	234	2.7%
`.../a[...]`	234	2.7%
`.../a`	117	1.3%
`.../@a`	24	2.7%
`.../text()`	11	0.1%
`.../a:*`	1	0.0%
nasty	97	1.1%

Figure 4.3: Our mining results on the select and match expressions in our gathered XSLT transformations. Each category covers the path shown, and any path similar to it in structure.

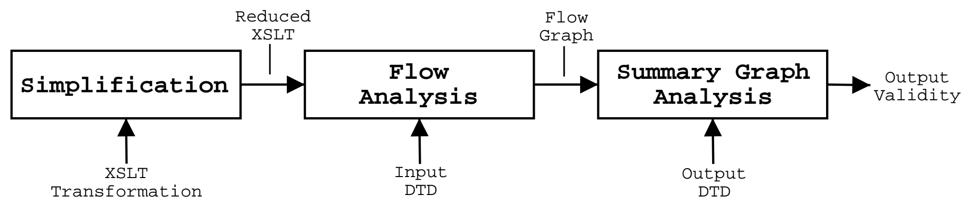


Figure 4.4: Overview of the static output validation analysis.

instructions: `apply-templates`, `for-each`, and `copy-of`, and the match expressions on template rules. The results can be found in Figure 4.3. They suggest that about 90% of all select and match expressions are simply dealt with, a further 9% ought to be reasonably approximated while the final 1% may pose serious problems for our analysis.

4.3 Overview

As Figure 4.4 illustrates, the analysis will be formed about three major phases:

- *Simplification* shall reduce XSLT documents to a much simpler core language,
- *Flow analysis* will analyze the XSLT transformation, trying to figure out the behavior of the tranformation, and the

- *Summary graph analysis* shall transform the results of the flow analysis into an output summary graph representing possible output of the transformation, and try to establish whether all output conforms to the output DTD.

In the following sections, each phase shall be described in detail.

4.4 Simplification

As with many other W3C standards, XSLT is a vastly complex language. It has many different constructs for doing the same thing, and there is lots of subtle details to be aware of in the specification. In order to simplify our work in the rest of the analysis, we start by performing a number of simplifications on the transformation being analyzed. Some similar ideas can be found in "Translating XSLT programs to Efficient SQL queries" [45].

The three phases of the simplification will proceed by applying a number of *simplification rules* in a specific order, each taking care of a single, or a group of similar constructs. The phases are:

- The *semantics preserving simplification* essentially rewrites redundant language constructs and normalizes those that are left by inserting default values. As the name implies, these simplifications make no semantic changes to the transformation, and could therefore be used in many other contexts than our static output validation. However, this first level of simplifications can only take us so far. The language is still complex after this phase. Note also that some of these simplifications rely on our limitation of no `namespace` axis (recall Section 3.4). They were in fact one of the main reasons for imposing this limitation in the first place.

- The second phase consists of *validity preserving simplifications*. This means the simplifications may introduce semantic changes, but only changes that does not affect the static output validation analysis.

- Lastly, a number of *approximating simplifications* are applied. These are basically an early filter, reducing constructs and features that are hard to analyze to less accurate constructs that we are able to analyze to some degree. In other words, we basically remove complex information we will not be analyzing anyway.

The result of the simplification will be transformations defined in a fragment of XSLT that we shall call *reduced XSLT*. The fragment is essentially formed around the core of XSLT: Template rules and `apply-templates` instructions. It will contain only the features which we will be analyzing in the following

sections. The difference from the related work is that we perform a conservatively approximating mapping from XSLT to reduced XSLT. This is a major step on the way towards practical usefulness.

Before we start on the actual simplification phase, we must check that no element in the XML templates, that are declared as `EMPTY`, contains any content other than whitespace. Theoretically, this is an approximation, since a non-empty content template of the given element may evaluate to empty. However, in practice no `EMPTY` declared element should contain any XSLT instructions. It should simply be empty.

The point of this empty check is that it allows us to ignore commments and processing instructions in the output, for the rest of the analysis. This will simplify or work significantly.

Semantics Preserving Simplifications

As explained above, the first group of rules preserve the semantics of the transformation. Therefore the stylesheet resulting from these simplifications does completely the same as before the simplifications.
The most important of the simplifications are:

- `import` and `include` resolution: Any external documents taking part in describing the transformation are merged into the document, taking proper care of *import precedence* with priorities and so on.
- Replace redundant constructs: `for-each`, `if`, `call-templates`, `copy-of` with node sets, and `copy` instructions are replaced with the equivalent constructs. This means a `choose` instruction instead of `if` and an `element` instruction instead of `copy`. The rest of the instructions can be imitated by template rules and `apply-templates` instructions.
- Template de-nesting: Nested templates are moved to their own template rules. This involves `when`, `otherwise`, and `with-param`. Each of these will have a single `apply-templates` instruction in a unique mode, in practice pointing to their former contents. The new template rules will inherit the match expression of their former containing rule.
- Defaults insertion: All default attribute values such as priorities are inserted, and the otherwise implicit build-in template rules are made explicit.
- Variable resolution: All variables are replaced by their definitions.
- Uniform XML construction: All literal result elements[2] are converted to element and attribute instructions, and plain text and the `text` instruction are converted to `value-of` instructions.

[2] *Literal result elements* are simply elements not in the XSLT namespace.

This concludes what can be done without starting to alter the transformation semantics. Note again that the above simplifications rely on the limitations made in Section 3.4. In particular, the `namespace` axis turned out to invalidate several of the above simplifications.

Validity Preserving Simplifications

This phase takes care of a couple of things not relevant to our particular analysis. The simplifications presented here do not affect the outcome of the analysis as a whole, but still serve to simplify the language being analyzed. Here, we perform the following:

- HTML to XML output method: There is a number of issues with handling the HTML output method itself, so in order to ease our job, we convert HTML output to XHTML output in XML mode instead. This means: (1) Converting all element and attribute names to lower case, (2) converting all `name` attributes to `id` attributes, (3) converting enumerated attribute type values to lower case, (4) setting the XHTML namespace, and finally (5) there is some small structural changes which can be expressed in a DTD. An examination of the differences between HTML4.01 and XHTML 1.0 can be found in the XHTML 1.0 specification [65]. Our job is made a lot easier by the simple fact that the XSLT document already is in XML format. Thus all the non-XML constructs in HTML, such as unbalanced tags, are already handled. This also includes more subtle differences like attribute values already having been forced to normalized form, and that text in `script` and `style` elements already is escaped. If some other HTML version than the default is specified, any further differences between the versions can be rectified.

- Removal of comments and processing instructions: Although the comments and processing instructions in the input document may have decisive influence on the output, for example if they are selected and result in template rule instantiations. However, in the case of comments and processing instructions on the output, they are unimportant for our analysis: Comments and processing instructions are not in themselves part of the XML they are imbedded in, and DTDs put no restrictions on the occurrence of either. Note that comments and processing instructions found in the transformation document are part of the transformation and not the output. For the output they must be constructed with `comment` and `processing-instruction` instructions. And so, these we remove.

Approximating Simplifications

Here, we simplify a number of constructs that we do not intend to handle in our analysis. This mostly involves arithmetic calculations and string manipulations, as well as irrelevant constructs such as the `message` instruction. We use a couple of extension functions to denote *unknown values*: `xslv:unknownBoolean()` models any boolean value, and `xslv:unknownString()` analoguely describes arbitrary strings.

It is important to note that the simplifications are conservative and sound approximations with respect to the analysis, i.e. they preserve the soundness and conservatism of the analysis as a whole. The approximations always simply "reduce the information" in the transformation to something easier to analyze. The important features of this simplification phase are:

- Complex constructed element or attribute names are reduced to simply `"xslv:unknownString()"`
- Complex templates describing attribute values in the `attribute` instructions are reduced to simply `<value-of select="xslv:unknownString()"/>`.
- `key` functions are replaced by "//M" where M is the match expression of the corresponding key declaration.
- Constructs that are irrelevant for our analysis are removed. This includes `message`, `fallback`, `output`, as well as others.

The XSLT transformation has now been reduced to the absolute core of XSLT. The core that most transformations already more or less use. Some constructs has been roughly approximated, but this mostly involves constructs that are inherently hard to analyze or that are used rarely.

To show the principle in the simplifications, let us examine the `for-each` instruction. It can be reduced to an `apply-templates` instruction and a template rule, without changing the semantics of the transformation, as follows:

```
<xsl:for-each select="s">
template
</xsl:for-each>
```

is reduced to:

```
<xsl:apply-templates select="s" mode="m">
sorts
with-params
</xsl:apply-templates>

<xsl:template match="child::node()|/|attribute::*" mode="m"
priority="0">
params
template
</xsl:template>
```

The mode is generated automatically so that it is unique in the transformation. Note also that the priority is insignificant, as only a single template rule exists in the given mode. However, reduced XSLT requires a priority to be defined. The `param` and `with-param` instructions forward any parameters defined in the template rule containing the `for-each`. This is necessary so that any references to the parameters inside the template continue to be defined with their proper values.

Reduced XSLT

What comes out of the last simplification phase, we call *reduced XSLT*. We summarize what is left in the language:

- Template rules, each with one match and one priority attribute, and possibly a mode and any number of template parameters.
- `apply-templates` instructions expressing all recursion in the transformation together with the template rules. The `apply-templates` instructions always have a select expression, may have a mode, and can be accompanied by any number of `sort` and `with-param` instructions.
- `element`, `attribute`, and `value-of` instructions handle all construction. Troublesome element and attribute names have been approximated away, and so has complex string manipulation for attribute values and text nodes.
- The `choose` instruction handle all branching in the transformations. We have not bothered trying to analyze the test expressions on each branch. They most likely do not have much influence on the output validity. Each branch in the `choose` has been moved to its own template rule.
- Those `copy-of` instructions with a single parameter reference that we could not resolve immediately or replace with other constructs have

```
      stylesheet ::= <stylesheet>toplevelelement*</stylesheet>
 toplevelelement ::= templaterule | param
    templaterule ::= <template match="pattern" priority="number" (mode="name")?>
                       (param*, xmltemplate)
                     </template>
     xmltemplate ::= instruction*
     instruction ::= applytemplates | choose | copyof | element | attribute | valueof
  applytemplates ::= <apply-templates select="nodesetexp" (mode="name")?>
                       (sort | withparam)*
                     </apply-templates>
          choose ::= <choose>(when+, otherwise)</choose>
            when ::= <when>applytemplates</when>
       otherwise ::= <otherwise>applytemplates</otherwise>
          copyof ::= <copy-of select="$name" />
         element ::= <element name="constructedname" >xmltemplate</element>
       attribute ::= <attribute name="constructedname" >valueof</attribute>
 constructedname ::= (name | {local-name()} | {xslv:unknownString()})
         valueof ::= <value-of select="valueofselect" />
   valueofselect ::= string | xslv:unknownString() |
                     string(self::node()) | string(attribute::a)
           param ::= <param name="name" select="exp" >xmltemplate</param>
       withparam ::= <with-param name="name" (select="exp")?>
                       xmltemplate
                     </with-param>
            sort ::= <sort/>
```

Figure 4.5: The grammar for reduced XSLT. *exp* refers to an XPath expression, *nodesetexp* refers to an XPath expression producing a node-set, while *pattern* refers to an XPath pattern. *name*, *string*, and *number* refers to the respective primitives of the XSLT specification.

been kept, so that parameter passing can be analyzed to some degree later in the analysis.

- Top level parameters describe completely unknown content. These are kept in order to examine if they are inserted somewhere in the output document. If they turn out to have direct influence on the output, we must signal an error. The output can then never be guaranteed valid.

A grammar for reduced XSLT can be found in Figure 4.5.

There are certain constructs in the reduced XSLT that we shall not be able to analyze. These are: (1) Template recursion on parameter values, and (2) Element and attribute construction with complex names now reduced to

`{xslv:unknownString()}`. Naturally we can never guarantee output validity in the presence of arbitrarily named elements or attributes. Our output DTDs can not express such content. Note that the `ANY` content model of the XML DTDs do not allow arbitrary element names. Regarding template recursion, we rely on our flow analysis for resolving template parameters, so when parameters themselves are used in defining the recursion, the problem becomes much harder. We do not attempt to handle such cases.

Either of the above two problem cases shall produce a suitable error report.

As an example of a simplification run, the news XSLT transformation from Figure 2.4 can be found in simplified form in Figure 4.6. Note that literal content has been kept intact, or the transformation would have been utterly unreadable to the human eye. This example far from covers all the intricacies of the simplification phase, but it nevertheless serves to show the general idea.

4.5 The Flow Analysis

We begin with some overall considerations regarding the design of the analysis. As described in the beginning of the chapter, we must figure out the "flow of construction" in the given XSLT transformation. This will allow us to construct a *summary graph* representing all possible output of the XSLT transformation. We can not expect to be able to figure out the flow perfectly, so the summary graph will most likely end up describing also output that can in fact never happen. However, the better we are at figuring out the flow of construction, the more accurate we will be able to make the summary graph. We call the process of figuring out the flow of construction for a *flow analysis*.

At this point, a number of approximations has already been done in the simplification phase, that will result in reduced accuracy of the result summary graph, but these mostly involve features which are very seldom used and hard to analyze. The worst approximations are already done at this point.

Recall that XSLT consists of a number of template rules and instructions. Most of the instructions are responsible for inserting XML fragments in their place during the transformation process. The flow analysis must basically locate all these "fragment-inserting" XSLT instructions, and then identify which fragments of XML can be inserted at each of these instructions.

The fragment-inserting instructions of reduced XSLT are: `apply-templates`, `value-of`, `choose`, `copy-of`, `element`, and `attribute`.

The `apply-templates` instructions must, as stated earlier, be connected to their possible target template rules. This is nothing less but the core of

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml"
                xmlns:xslv="urn:XSLTValidator">
  <xsl:template match="child::news" priority="0.0">
    <html>
      <head>
        <title>News</title>
        <meta http-equiv="Pragma" content="no-cache" />
      </head>
      <body>
        <xsl:apply-templates select="child::item" />
      </body>
    </html>
  </xsl:template>

  <xsl:template match="child::item" priority="0.0">
    <xsl:apply-templates select="child::headline" />

    <font face="arial,helvetica" color="#808080" size="1">
      Den
      <xsl:apply-templates select="attribute::date" />
      Kl.
      <xsl:apply-templates select="attribute::time" />
    </font>
    <br />

    <xsl:apply-templates select="child::text" />

    <hr />
  </xsl:template>

  <xsl:template match="child::headline" priority="0.0">
    <font face="arial,helvetica" size="4">
      <b>
        <xsl:value-of select="string(xslv:unknownString())" />
      </b>
    </font>
    <br />
  </xsl:template>

  <xsl:template match="child::text" priority="0.0">
    <b><xsl:value-of select="string(xslv:unknownString())" /></b> -
    <xsl:apply-templates select="child::p" />
  </xsl:template>

  <xsl:template match="child::p[xslv:unknownBoolean()]" priority="0.5">
    <xsl:value-of select="string(xslv:unknownString())" />
  </xsl:template>

  <xsl:template match="child::p" priority="0.0">
    <p>
    <xsl:value-of select="string(xslv:unknownString())" />
    </p>
  </xsl:template>

  <!--BUILT-IN TEMPLATE RULES-->
  <xsl:template match="child::*|/" priority="-0.5">
    <xsl:apply-templates select="child::node()" />
  </xsl:template>
  <xsl:template match="child::text()|attribute::*" priority="-0.5">
    <xsl:value-of select="self:node()" />
  </xsl:template>
  <xsl:template match="child::processing-instruction()|child::comment()" priority="-0.5" />
</xsl:stylesheet>
```

Figure 4.6: The news example of Figure 2.4, after simplification has been performed. Literal content has been preserved in order to make it more readable.

XSLT processing. Even more so after the simplification phase, as many language constructs of XSLT has been reduced to templare rules and `apply-templates` instructions. As we shall see, it will also be the hardest nut to crack in the flow analysis.

`value-of`, `element`, and `attribute` need no connecting, as they contain all necessary information already. They may depend on the containing XML template for resolving "`local-name()`" expressions, but that is for the containing template rule to resolve.

Finally the `choose` instruction contains a number of possible *branches* in the form of `when` and `otherwise` instructions. They each describe one possible XML template to be instantiated and inserted instead of the `choose`. Post-simplification, each of these branches each contain a single `apply-templates` instruction, and the analysis of `choose` will therefore depend entirely on the analysis of its descendant `apply-templates` instructions.

In recapitulation: All the XSLT instructions in reduced XSLT depend on the template rules and the `apply-templates` instructions for their analysis.

Now this connecting is obviously taking shape as a kind of graph, where the nodes are template rules, and the edges connect `apply-templates` selections with template rules. However, note that the edges originate from *within* the nodes of the graph. We shall call this an *XSLT flow graph.*

The principles behind our flow analysis and flow graphs are much the same as for the *Refined-TAG* graphs in "Static Analysis of XSLT Programs" [32], the *QTrees* of "Translating XSLT programs to Efficient SQL queries" [45], and the *execution flow trees* of iXSLT [81]. However, our end goal is not quite the same, and as we shall see, our flow analysis will go into more detail than theirs.

Some Problems to Consider

Before we go on, we should perhaps take a look at the select and match expressions which lie at the core of the template rules and `apply-templates` instructions: The first level of complexity in the select and match expressions are node-set unions. Both expressions can be a union of any number of location paths. This looks like it will lead to individual analysis of each select-match pair, so let us make the analysis simpler from the start by making the flow graph edges point between the individual location paths instead. This has to some degree already been achieved on the match side through template rule "splitting", but there still remains unions in the matches. It is simply not always semantically sound to split template rules: If the two match paths *overlap*—i.e. are able to match on some of the same nodes—and if they end up with the same priority, they can represent two alternative template rule targets for the same node in a selection. This is illegal in the processing model of XSLT. Neither is it semantically sound to split `apply-templates`

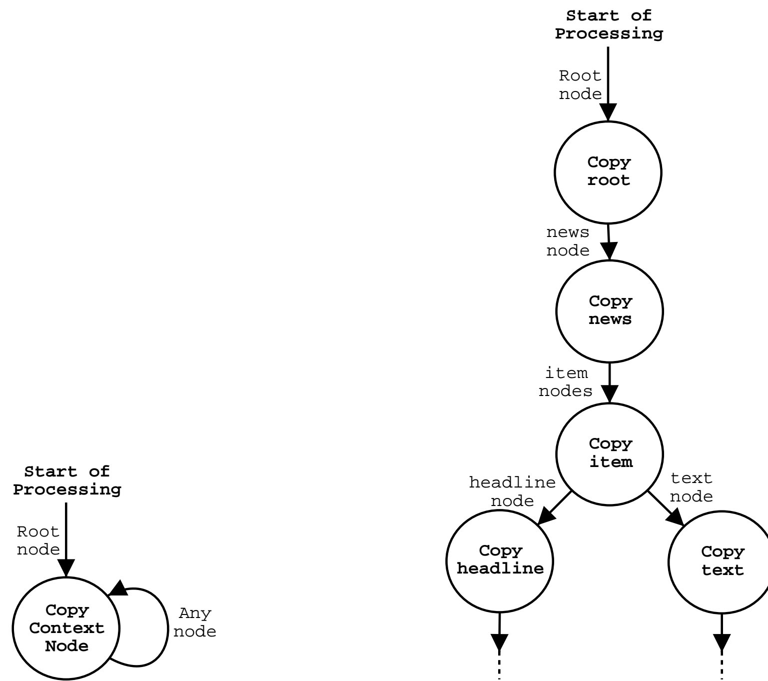


Figure 4.7: Imprecise flow graph for the *general identity* transformation.

Figure 4.8: More precise flow graph for the *general identity* transformation, mirroring the input DTD, which in this case is the news DTD of Figure 2.2.

select paths. The document order of the selections must be preserved.

This now means that our flow graph will describe what templates can be instantiated by nodes selected by each path in a selection separately, and it will describe what match path it matched on.

But, already we run into trouble. One of our main goals for the analysis is to be able to analyze the general identity transformation (See Figure 3.1 in Section 3.2). If we connect the `apply-templates` instruction as described above, we get that it eventually can and will select every node in the input document, since the template rule will be instantiated with every possible type of node on the input as context node. Figure 4.7 visualizes the flow graph this will result in.

We see that the connections of the graph indicate that any node can be selected by the `apply-templates` instruction of the template rule. Each select, match and following instantiation (Recall Section 2.7) of the template leads to a simple copy of the context node followed by recursion. Now, the transformation process starts with the root node, and—according to the flow graph—proceeds to copy arbitrary nodes in arbitrary structure. This

is indeed a rough approximation, given that we know exactly how the input documents look. The transformation process will in practice recursively copy children of the input, eventually producing an exact copy of the input. So the flow graph ought to simply mirror the structure of the input schema, as illustrated in Figure 4.8. A very similar situation of "arbitrary recursion" occurs when the built-in template rules are used. They do not copy the nodes as the general identity template rule does, but they recurse on children in the same "minimal information" way as the general identity. Also, nothing hinders the user himself in producing similarly troublesome template rules.

This is not the only problem. The `copy` instruction (Converted to an `element` instruction in reduced XSLT, but it is the same thing), used for example in the general identity, is very poorly supported in the output summary graph so far. If template rules are represented in a single summary graph node for each rule, the `copy` instructions will have to point to all possible nodes for which the containing template rule can be instantiated. This turns out horrible in the general identity: It will end up pointing to all possible types of nodes on the input, for insertion. But the problem will occur also in much simpler cases. The `copy` instruction makes little sense to use unless there are more than one possible context node, so this can be expected.

Finally, we have reduced the *copy-of* instruction to *appy-templates* instructions and template rules, utilizing a structure similar to the general identity. This will be equally hard to analyze.

In light of these things, we have to do something if we wish to be able to analyze these constructs, and well as meet our goals.

A More Detailed Design

Our solution to the problems presented above shall be to create a summary graph expressing the flow of context nodes more accurately, and to extend the flow analysis to recognize these flows and make proper use of the more detailed summary graph. We change our analysis from being *monovariant* in the template rules to being *polyvariant*. In other words: We will now not only create a single summary graph node for each template rule, but one for each of the possible context nodes the template can be instantiated with. This means that the template in each of these summary graph nodes represent instantiation of a template rule with a specific context node. A `copy` instruction can now simply be replaced with the context node, and `apply-templates` selections can take the context node into account.

To support the detailed summary graph model, we extend the flow analysis by making it "context-flow sensitive". That is: We will analyze how each context node flows, by examining each template rule for each in-flowing context node, and determine, as precisely as we can, what flow each of the

contained `apply-templates` instructions are responsible for.

These improvements give us the power to express for example the output of the general identity, mirroring the input DTD: The template rule will result in an summary graph node for each node type in the input class, making the `copy` a simple matter, and the child recursion can be correctly expressed by referring from the `apply-templates` to the possible children of the context node.

A potential problem with the above more detailed model is the blow-up in complexity: The increased amount of times each template rule must be analyzed, and the increased number of summary graph nodes. Luckily however, there seldom are more than one possible context node for a template rule in practice, except of course from the trouble-cases mentioned earlier. But certainly, some rules matching on more than a few nodes can blow up the complexity considerably. The same goes for many different *modes* each making use of the built-in template rules. However, we do get a considerably improved precision, so the price definitely seems worth paying. We will examine the cost of these blow-ups in Section 5.4.

Notation

Let us define a notation for referring more conveniently to the entities involved in the flow analysis. To start, let T denote the transformation being analyzed *after simplification*. This phase will work only on the simplified transformation, so we will assume that the transformation has been through the simplification phase. Then let D_{in} and D_{out} represent the input and output DTDs respectively. As we will be working a lot with the information contained in the input DTD, let further $\mathcal{E}$ contain all element names from D_{in}, and similarly $\mathcal{A}$ shall contain the attribute names. For some other DTD D, we shall write $\mathcal{E}_D$ and $\mathcal{A}_D$
Now, let Σ refer to the *node types* of D_{in}, and be defined as follows:

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

where **root**, **pcdata**, **comment**, and **pi** respectively represent the root node, text nodes, comment nodes, and processing instruction nodes.

This set Σ describes the node types distinguishable under the XPath data model (recall Section 2.5), which we have adopted, with the exception that the names of processing instructions are not differentiated. Processing instructions are not part of the XML itself, and our output schema model (DTD) is not able to constrain the occurrence of processing instructions, or their names. And add to this, that processing instructions are hardly ever used. Comments see more use, but seldom have influence on the output of XSLT transformations. Nevertheless, selection of comments and processing instructions from the input shall be modeled, albeit ignoring the names of

the latter.

Note that we shall make no distinction between the *names* of elements and their *node types*, i.e. $\mathcal{E}$ shall be used interchangably to refer to the names and to their corresponding node types. Also, the coupling of attributes with their element is natural, since attributes with the same name, but with different parent elements, essentially are two different attributes. They each have their own definition in the attribute list of their parent element, defining various properties, including allowed values. We shall write $name(n)$ to refer to the name of $n \in \Sigma$:

$name : \Sigma \mapsto \mathbb{S}$, where $\mathbb{S}$ refers the set of all strings.

$$name(n) = \begin{cases} n & \text{if } n \in \mathcal{E} \\ a & \text{if } n = (a, e) \in \mathcal{A} \times \mathcal{E} \end{cases}$$

Although DTDs do not in themselves allow constraining which elements are allowed as the *document element*, we shall incorporate this information in the analysis. When the knowledge of allowed document elements is present, let the set of those elements be denoted as: $\widehat{\mathcal{E}} \subseteq \mathcal{E}$. If no such knowledge exists, we must assume that any element is allowed as document element, and we let: $\widehat{\mathcal{E}} = \mathcal{E}$.

Having the DTDs in place, let now the template rules of T be denoted $\mathcal{T}$. Furthermore let the following functions refer to the match expression, mode, and priority of a template rule respectively:

$$match : \mathcal{T} \mapsto \Phi_{pattern}$$

$$mode : \mathcal{T} \mapsto \mathbb{S}$$

$$priority : \mathcal{T} \mapsto \mathbb{R}$$

Here, Φ is the set of all XPath node-set expressions, and $\Phi_{pattern} \subset \Phi$ refers to the subset that is patterns, as defined in the XSLT specification [23].

Extended Content Models

There is an inherent complexity for our analysis in the sharp lines that are drawn between the different node types of the XPath data model. The elements are described by content models in the DTD, the attributes by the attribute lists, the root content is defined by the XML standard, coupled with possibly the knowledge of a specific document element. Apart from this, comment and processing instruction nodes can—according to the XML specification—appear anywhere in a document. We shall unify all this by

extending the content models of our input DTD to general regular expressions[3] over the node types in Σ, and by defining a content model for each of the node types in Σ. For referring to these extended content models we shall write: $contentmodel : \Sigma \mapsto REG_\Sigma$, where REG_Σ are all regular expressions over the alphabet Σ. We write *sequences* of content in parantheses and single nodes without, such that for example the following holds: $(\mathsf{a} \cdot \mathsf{b}) \in exp$, where $exp = (\mathsf{a}, \mathsf{b}^*)$, and: $\mathsf{b} \in exp$. The latter statement simply expresses that b nodes occur in *exp*, or in other words, that b nodes can occur in the sequences described by *exp*. The former statement describes one such sequence of *exp*.

The *extended content models* shall be defined as follows:

- For elements, the content model of the DTD is expanded with $(\mathbf{comment}|\mathbf{pi})^*$ on both sides of every name in the original content model. Thus the content model for $\mathsf{item} : (\mathsf{headline}, \mathsf{text})$ of the DTD in Figure 2.2 is expanded rather verbosely to: $(((\mathbf{comment}|\mathbf{pi})^*, \mathsf{headline}, (\mathbf{comment}|\mathbf{pi})^*), ((\mathbf{comment}|\mathbf{pi})^*, \mathsf{text}, (\mathbf{comment}|\mathbf{pi})^*))$. The expansion is only linear in the size of the content model though.
  In the case of ANY, the content model becomes $(\Sigma - \mathcal{A} \times \mathcal{E})^*$, and with EMPTY it shall be the empty string $()$.

- At the top level, only one of the elements from $\widehat{\mathcal{E}}$ is allowed, but also any number of comments and processing instructions. Thus, the content model of the root shall be: $((\mathbf{comment}|\mathbf{pi})^*, \widehat{\mathcal{E}}, (\mathbf{comment}|\mathbf{pi})^*)$.

- For the rest of the node types, i.e. attributes, text, comments, and processing instructions, the content model shall be the empty string $()$ as well. None of these nodes can have any children.

Note that attributes are still not part of the content models of elements. Their unorderedness make them fundamentally different from the rest of the content. Merging them with the content models pose some inherent difficulties, as their order in the expressions should be ignored, and they cannot have any cardinality except a single occurrence.

4.5.1 The XSLT Flow Graph

Let us define more precisely what the structure and semantics of the flow graph shall be.

[3] As opposed to the deterministic regular expressions of DTDs. See the XML specification [18].

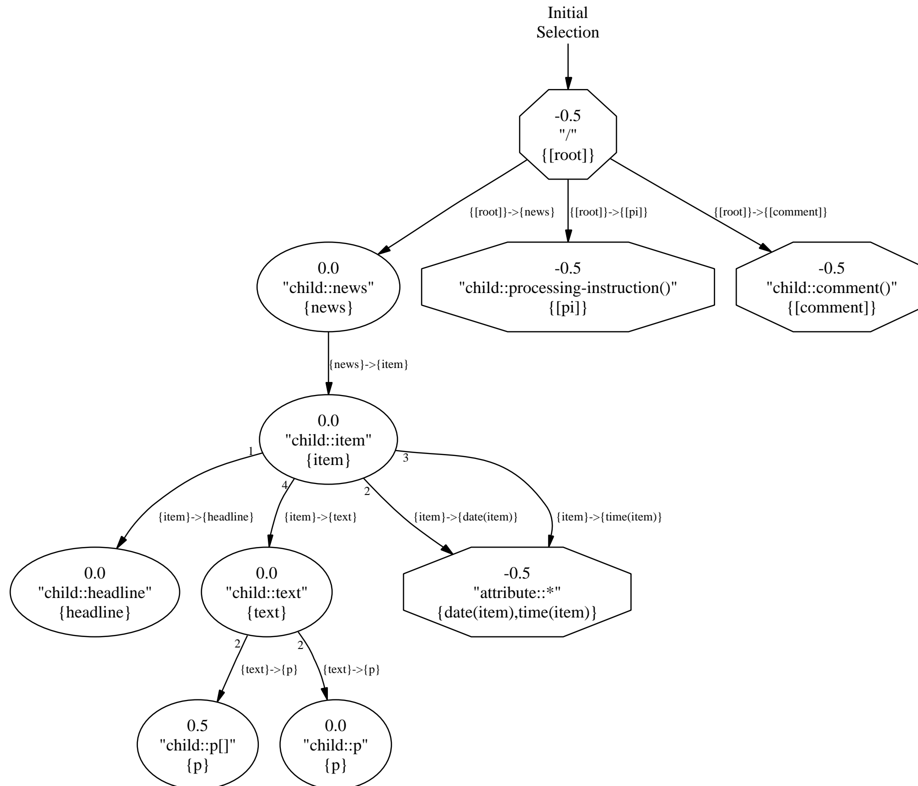


Figure 4.9: Visualization of the flow graph for the simplified news transformation described in Figure 4.6. The numbers on the tails of the edges denote which XSLT instruction it originates from, and the octagon shaped nodes are the built-in template rules. Each flow graph node is labeled with its priority, match pattern, and possible nodes for instantiation. The edges are annotated with which instantiations produce what flow. Thus, `{news}->{item}` means that when the template is instantiated with a `news` element, an `item` element will flow along the edge. The special node types have been put in square brackets, such as `[root]`.

The Nodes

We do not need to mirror the nodes of the summary graph in the flow graph. Having a flow graph node for each possible context node will do nothing but complicate the graph unnecessarily. Instead, we shall annotate the nodes and edges with the knowledge of the flow of context nodes. This construction will express exactly the same as individual graph nodes for each context node, but the edge annotation is a much more compact representation.

We will however have a separate flow graph node for each match location path in the case of a union. As we shall see in a moment, this will improve the analysis precision slightly.

In order to have some idea of what our goal is, Figure 4.9 illustrates the flow graph of our running example. Each node is labeled with: (1) Its priority, (2) its match patttern, and (3) the set of possible nodes for instantiation. As can be seen, this graph describes accurately what types of nodes each template rule can be instantiated with. Also, each edge is annotated with the index of the XSLT instruction it originates from, and the flow that it is responsible for. The diamond shaped nodes in the illustration represent the built-in template rules. In this example, the built-in template rules take on the two typical jobs that the built-in template rules are responsible for: (1) "Filling in" with simple downwards child recursion when no explicit rules are provided, and (2) copying text over directly from attribute values or text nodes.

When the time comes for summary graph construction, we need to know what possible context nodes a template rule will be instantiated with. To support this, we will attach a *context set* to each flow graph node. Initially, these sets will be empty. One of the jobs of the flow analysis is then to fill the context sets as tightly around the sets of actually possible context nodes. Note that some information is in fact lost in this process. For example, a template rule with the match expression `"child::b/child::c"` will be instantiated only with `c` nodes that have an `b` node as parent. But the context set will contain simply a `c` node. In other words: Any knowledge about the context node, apart from its node type and possibly name, is lost. The information about the parent `b` in this case is still present through the match expression, but what if the match expression had just been `"child::c"`, and the only selection pointing to our template rule being `"child::b/child::c"`? The information about the parent now lies one step away in the flow graph, at the source selection. In general, the context information not approximated away could always be extracted by examining all the flow eventually ending up in the template rule in question, or one could try and propagate it with the flow, by describing the context with for example regular tree automata. However, such precision in context knowledge is complicated, and it is hardly necessary. As we shall see, we will come quite far with simply the context sets.

We can now see the benefit of separate flow graph nodes for each match path. The analysis of instructions inside the given template rule can benefit from the specific match location path. In other words: We maintain the knowledge about which match path the node matches on in the further analysis. It also simplifies the analysis of those instructions, by having only a single match pattern location path to consider.
In a more concise form:

> *That the node σ lies in the context set of some flow graph node N means that the template rule that N represents possibly can*

be instantiated with σ as context node. That σ does NOT lie in the context set means that the template rule of N never can be instantiated with σ as context node.

The conservatism in the semantics of the flow graph must support the conservatism of the entire analysis: *If some event is possible for an actual run of the transformation on some input conforming to the input DTD, then the event must be expressed in the flow graph.* Concretely, in the case of the context sets, this means that all nodes that some template rule can be instantiated with as context nodes, must be present in the context set of at least one of the flow graph nodes representing the template rule. This is just another way of saying what is stated more formally above. Of course, the statement about the conservatism of the flow graph must hold in all aspects of the graph.

The Edges

The edges of the flow graph shall—as discussed—point from individual location paths in `apply-templates` selections, to possible target templates. In order for the now polyvariant summary graph construction to accurately connect the nodes of the summary graph, we need knowledge of the flow of context nodes along the edges of the flow graph. Thus, each edge in the flow graph shall describe the *edge flow* that the edge produces. The edge flow maps each context node of the containing flow graph node to a set of context nodes. This set describes the context nodes that can end up instantiating the target template rule, and which matches the targer match location path (Remember, edges point to individual location paths in the match expression).
More precisely:

An edge from the selection S in the flow graph node N_{from}, to the flow graph node N_{to}, means that for some context node in the context set of N_{from}, it is possible that S selects some node in the context set of N_{to} which may instantiate the template rule which N_{to} represents. Absence of the edge means that the template rule that N_{to} represents can never *be instantiated with nodes selected by S.*

This also means that if there is no edges from any of the selections of some `apply-templates` instruction to any of the flow graph nodes representing some template rule, then that template rule can never be instantiated at that `apply-templates` instruction.
Regarding the edge flow, we have that:

Given an edge between (N_{from}, S) and (N_{to}, M), an edge flow with the context node σ, to the set of node types F, means that

> *each node in F possibly can be selected by S with σ as context node, and that each node in F possibly can be matched by M. Each node NOT in F can never both be selected by S under σ and matched by M.*

Note that a node in the edge flow implies that this node must be in the context set of the target flow graph node.

The flow graph initially only contains the nodes as defined above. The nodes of the graph are fixed, although if their context set after the flow analysis is empty, they might as well not be in the graph, as they can never be instantiated. This is not to be viewed as an error in itself, as for example the built-in template rules often are not used.

Also, the context set of each flow graph node is empty, and there are no edges in the graph. This shall serve as the starting point for the flow analysis.

Formally

We shall now formulate the structure of the flow graph algebraically. Not so much for the notation it provides, but rather we shall use it as an descriptory tool, describing more concisely what we have said with words.

Let the set of all `apply-templates` *selection paths* in T be denoted $\mathcal{P}_s$, and similarly the match paths of the template rules in T shall be $\mathcal{P}_m$. By *paths*, we refer to each location path in the select and match expressions separately. Also, each separate selection or match path in the transformation document shall correspond to an element in either $\mathcal{P}_s$ or $\mathcal{P}_m$, so several of the same location paths may be in a set, so long as they each correspond to a different piece of the transformation document. Let also N be the set of all flow graph nodes. Note that there exists a one-to-one mapping between the flow graph nodes and the match paths.

The XSLT flow graph shall now be defined as:

$$XFG = (\widehat{N}, E, \mathcal{C}, \mathcal{F})$$

Where:

$\widehat{N} \subseteq N$ are the *root nodes* of the graph,
$E \subseteq N \times \mathcal{P}_s \times N$ are the *edges* of the graph,
$\mathcal{C} : N \mapsto 2^{\Sigma}$ describes the *context set* for each node, and
$\mathcal{F} : E \times \Sigma \mapsto 2^{\Sigma}$ maps each edge to its *edge flow*.

We further define the following mappings:

$rule : N \mapsto \mathcal{T}$ maps a flow graph node to the template rule from which it originates.
$target : E \mapsto N$ describes the target graph node of an edge.
$match : E \mapsto \Phi$
$select : E \mapsto \Phi$
$priority : E \mapsto \mathbb{R}$ assigns a priority to each edge.

Note that the priority of an edge is simply the priority of the target graph node's template rule (i.e. $priority(e) = priority(rule(target(e)))$).

We shall refer to the target flow graph node, and the template rule of the target flow graph node interchangably when it is not of significance. Likewise we shall not distinguish between an edge and its target unless it is relevant.

4.5.2 The XSLT Flow Analysis

Starting out with the initial flow graph described in the previous section, we must now fill in the graph so that it respects the semantics we have established. This means (1) filling the context sets of the graph nodes, (2) creating the graph edges, and (3) annotating the edges with the edge flows. We shall accomplish this by starting with the root node, and *propagating the flow* along the edges, until all possible flow has been described. This process of flow propagation is in a way a simulation of the transformation process on all possible input trees at once.
More specifically, we shall need the following parts for the analysis:

- A *flow propagation algorithm*, describing how flow moves along the edges.
- Some *flow propagation tests*, conservatively approximating the possible flow along some possible edge such that the semantics of the flow graph is upheld.
- A *priority override filter*, removing edges which, because of its priority, never can carry any flow. This is of paramount importance for ruling out the always present built-in template rules.

The flow propagation algorithm is the foundation of the flow graph contruction. It describes how the propagation tests and the priority override filter is used. We shall describe this first.

The Propagation Algorithm

The XSLT transformation process starts by finding all template rules with no mode, that can match on the root node. A single target template rule for

the node must be found, the same as any other instantiation, and that rule is then instantiated to form the root of the output document. Similarly we must initiate the flow propagation by finding out with what template rule the process starts. The idea can be nicely described through a small piece of XSLT that we shall call the *initial selection*:

```
<apply-templates select="/"/>
```

This `apply-templates` instruction perfectly describes the initiation process. To start the flow analysis, we simply start by finding out which template rule can match on this initial selection of the root. Note that the `apply-templates` instruction above serves exclusively to illustrate the mechanism. It is neither a part of the XSLT transformation nor of the flow graph.

Now, the flow propagation algorithm centers around analyzing the implications of a new node being added to a context set. The establishment of this new node as a possible context node for instantiation, implies that the `apply-templates` selections in the corresponding template rule possibly can select new elements in the context of this new context node. Thus, we need to establish what nodes all the selection paths in the rule can select under this context node, and what template rules are possible targets for these selected nodes. The resulting flow graph edges must further be annotated with their possible edge flow. And if this means that new nodes flow into some flow graph nodes, then these graph nodes must in turn be analyzed in the same way, under the new context nodes added. Whether the flow propagation proceeds in breadth-first or depth-first manner is not of importance.
The flow analysis proceeds as follows:

1. For each selection path p in flow graph node, iterate through each flow graph node n with the same mode as the of p enclosing `apply-templates` instruction and calculate the possible flow from p to n. This flow is calculated through intersecting the results of the individual flow propagation tests. Their consevatism ensures that the nodes which can flow along an edge in practis, always will be in the edge flow sets.

2. For each pair of edges we examine if one of the edges overrides flow in the other edge. I.e. we examine if some nodes will always flow to the higher priority target. This is the priority override filter.

3. Any edge flow established is now fixed, and will remain unchanged throughout the rest of the flow analysis. Any edge flow resulting in context nodes that were not already there cause a recursion of this analysis on the flow graph node and the new context node.

It is important to note that the flow analysis for a given context node in no way interferes with the edge flow already established for the other nodes in the context set. The point of emphasizing the static nature of the flow for a given context node is, that at the point when the flow has been propagated out along the edges to other graph nodes, it cannot simply be recalled: The propagated flow may be responsible for any number of flow propagations further away in the graph. Because the priority override filter is able to remove flow, it is imperative that we finish the priority override filtering, before adding the edge flow to the context set of the target graph node and the subsequent recursion. Since priority override only examines the edge flow for a specific context node, and on two edges out of the same selection path, we simply need to finish analyzing each selection path for the given context node before recursing. This is accomplished in the flow propagation algorithm described above.

In the context of the above algorithm, the flow propagation analysis is initiated by analyzing the possible edge flow, out of the initial selection, as if a new context node had just been added to the selection's flow graph node (although it has none). As we can see, the initial selection is entirely independent of context nodes and match expressions. It always just selects the root node.

The Flow Propagation Tests

The flow graph must describe possible nodes to flow along an edge. In other words: It describes what nodes can be selected by the given `apply-templates` instruction, when instantiated with a certain context node. As explained, the analysis is done on the individual selection paths of an `apply-templates` instruction.

Since the `apply-templates` instruction lies in a certain template rule, we also know that the context node matches one of the match paths in the match pattern of the template rule. Again, since our flow graph contains nodes for each match path, a specific match path is given. In other words, we are given the following four facts when analyzing some `apply-templates` selection:

(1) A *source match path*,
(2) an *instantiation context node* matching the match path,
(3) a *selection path*, and
(4) a target flow graph node with its attached *target match path*.

Given these four pieces of information, *we must conservatively construct a set of context nodes such that nodes* not *in the set is guaranteed either (1) to not match the target match path, or (2) not to be selectable with the selection path, from a node that both matches the source match path and that is of*

type equal to the context node. This set is to be the edge flow annotation of the relevant edge, and we shall call an algorithm determining such sets for a *flow propagation test*. In the construction of these edge flow sets, the conservatism means that the sets always must describe "too much" flow.

The problem is now essentially reduced to examining the *compatibility* of the selection path with the target match path, under the input DTD. For example, nodes selected by the path "`child::a/child::b`" could never match the pattern "`child::c/child::b`". Only `b` nodes with `a` parents are selected, while only `b` nodes with `c` parents match the pattern. We say that these two paths are *incompatible*.

Taking a peek at the four pieces of knowledge available to us, we can see that the selection path will start the selection process from the context node, and that this context node is guaranteed to match the source match pattern. This allows us to do a more precise compatibility test by expressing all of these three things in a single location path. The source match and selection paths can simply be concatenated, but the knowledge of the context node should be included too. If for example the match path is "`child::*`", and the context node an `a` element, we can express both pieces of knowledge as "`child::a`". One might be tempted to simply express the knowledge of the context node in a `self` axis expression such as "`self::a`", and then concatenate. The above example would become "`child::*/self::a`". In this case it works, but since an XPath name test only leaves nodes with a matching name that are also of the principal node type—which for the `self` axis is elements—it does not work if the context node is an attribute.

In general, we know that the context node matches the source match path. Thus we know that the context node matches the node test of the last location step[4] in the source match path. It is then simply a matter of—in a way—intersecting this last name step with the context node type. We elaborate:

Let M_s be the source match path, c the context node, and S the selection path. The concatenation of these is defined as:

$$CONCAT(M_s, c, S) = \begin{cases} S & \text{if } S \text{ is an absolute path} \\ /S & \text{if } c = \textbf{root} \\ MERGE(M_s, c)/S & \text{otherwise} \end{cases}$$

$$MERGE(P\ \mathit{axis::test}[P_1] \cdots [P_n], c) = P\ \mathit{axis::}NODETEST(c)[P_1] \cdots [P_n]$$

$$NODETEST(c) = \begin{cases} \mathit{name}(c) & \text{if } c \in \mathcal{E} \vee c \in \mathcal{A} \times \mathcal{E} \\ \texttt{text()} & \text{if } c = \textbf{pcdata} \\ \texttt{comment()} & \text{if } c = \textbf{comment} \\ \texttt{processing-instruction()} & \text{if } c = \textbf{pi} \end{cases}$$

[4]The *last location step* in a location path refers to the rightmost step. This stems from the processing model for evaluation of XPath location paths [25].

This construction is similar to the location path concatenation of Dong and Bailey in [32], except that we have further incorporated the context node.

An important fact for the soundness of the above is that the axis of the last location step in M_s always will be either `child` or `attribute`: This is due to the limited syntax for match patterns (recall Section 2.7). A match pattern is only allowed to used the `child` and `attribute` axes, as well as the `//` construct, which is syntactic sugar for `"/descendant-or-self::node()/"`. This `//` construct is however only allowed in the beginning of the location path, or as separator between two normal location steps. Thus, a pattern can never end with a step using the `descendant-or-self` axis.

The reason for all this is that the MERGE operation above is not sound if the axis of the last step in the pattern is able to select both elements and attributes. We can not make a test on the name of both elements and attributes at the same time, without resorting to predicates. And since we generally avoid predicates in our analysis, we do not want to create more than there already is. But as argued above, the source match path—being an XPath pattern—can only end with either the `child` or the `attribute` axis. The former can not select attributes, and the latter can only select attributes. Thus the MERGE operation will always be sound.

Because of the way we define attributes of each element separately, a small improvement on the path concatenation can be made in case the context node is an attribute and match path a single attribute step. Since the context node expresses the name of its parent element, we can further elaborate the match path by prefixing it with a `child` axis step with the given parent element name.

As an example of constructing a concatenated selection path, assume that M_s = `"child::*"`, c = `a`, and S = `"child::b"`. Then we get:

$$
\begin{aligned}
&CONCAT(\texttt{child::*},\, \texttt{a}, \texttt{child::b}) = \\
&MERGE(\texttt{child::*},\, \texttt{a})/\texttt{child::b} = \\
&\texttt{child::}NODETEST(\texttt{a})/\texttt{child::b} = \\
&\texttt{child::a/child::b}
\end{aligned}
$$

Now that the flow propagation test has been reduced to a seemingly simple comparison of two location paths, let us examine the compatibility problem. Certainly, in the case of simple `child` axis steps and name tests, there is a simple linear algorithm: Run through the path from right to left[5] and check if the name tests agree. The edge flow consists simply of the element represented in the last name test.

However, when the name tests start using `*` and `node()`, it is no longer

[5] The paths may be of differing length, so we must start at the right.

trivial to find the edge flow of the selection, and testing compatibility with the target match path starts to depend on the DTD. Is for example "`child::*/child::*/child::b`" compatible with "`child::c/child::*/child::*`"? In themselves, it certainly is possible that they can select the same nodes, but it also depends on whether `b` elements can be children of children of `c` elements. In other words: It depends on the input DTD. And it does not get any easier if we start looking at the rest of the XPath axes.

In conclusion, there seems to be no simple and general analysis of this compatibility problem. We shall present two flow propagation tests: In Section 4.5.3 we present the *path simulation test*, while Section 4.5.4 takes another angle in the *paths automata test*. As we shall see, each propagation test strategy shall eliminate different kinds of nodes. We know that *if a node is NOT in the context set of one of the propagation tests, then it is not possible as a context node for instantiation, ever.* Thus, the final edge flow for an edge shall be the intersection of the results of the individual tests.

A direct consequence of this is that we might not have to perform more than one of the propagation tests. As it turns out, most *candidate edges* being examined in the flow propagation will be false, i.e. they can never propagate any flow. This falseness is often quite obvious and easily determined. And if just one of the flow propagation tests return the empty set as edge flow, then no further tests have to be made: It is already guaranteed that no nodes can flow along this edge. This also means that it makes sense to perform the fastest of the flow propagation tests first.

The Priority Override Filter

While the edge tests look exclusively at one edge, the edge filter examines edges from the same source in relation to each other. Each edge represents some possible flow to its target flow graph node. But so far, the priorities of the template rules have been completely ignored. This for example means that the built-in template rules always will be possible targets, because of their unrestrictive match patterns. But we know that they implicitly have the transformation-wide lowest priority[6]. Thus we need to handle the template rule priorities, if we wish any kind of precision in the flow graph.

The idea in the priority override filter is to examine each pair of edges going out of some selection path, under the context node currently being examined (recall the flow propagation algorithm described above). If one of the edges has higher priority than the other, it will, at runtime, always carry flow that it matches before the lower priority edge. However, statically we can not just remove any nodes in the higher priority edge flow from the lower priority one: Since the edge flows are conservative approximations, none of the nodes in the set are guaranteed to actually flow along that edge.

[6] made explicit in the simplification phase. Recall Section 4.4.

Furthermore, a node type being in some edge flow does not mean that *all* nodes of that type can flow along the edge. It might for example only be nodes with a certain named parent element.

So, we need an algorithm that is able to determine whether all nodes of a certain type, able to flow into the lower priority flow graph node, also always can match the higher priority graph node. In order to do this, we shall need some tools developed for the flow propagation tests. We will present such an algorithm in Section 4.5.5. It will naturally be a conservative approximation, but it turns out to be pretty accurate.

4.5.3 Path Simulation Test

One of our flow propagation tests (recall Section 4.5.2) is the *path simulation test*. The idea behind this particular test is quite simply to simulate the process of selection of an XPath location path. Not on an XML tree as is normally done, but on a DTD. In particular, we simulate the execution of the concatenated source match path, context node, and selection path described in Section 4.5.2 on the input DTD. This will give us a set of node types that are guaranteed to cover the types which can be selected by the path in practice. We then run a similar simulation on the target match path being analyzed, in order to get a set of nodes describing the possible nodes which the target match path can match on. The intersection of these two sets shall cover the node types which can be both selected by the concatenated selection path and which match the target match path. Since we work on sets of node types from Σ, this test is oblivious of context information apart from the node types, but it can handle all the axes, except the `namespace` axis, which we have left entirely out of our treatment. Under a model that included namespace nodes, this simulation test ought to be easily extended with the `namespace` axis.

As an example of the simulation test, consider the concatenated selection path `"child::a/child::b"` and the target match expression `"child::b"`. The path simulation will yield the set {`b`} for both paths, stating correctly that `b` nodes can flow along this edge. Now, if the target match path had instead been `"child::q/child::b"`, the simulation would still yeild the same {`b`} sets for both paths, but nodes selected by the concatenated selection can in fact never also match the target match path. This illustrates the lack of information about the context of the nodes in the simulation sets. As we shall see in Section 4.5.4, the *paths automaton test* will be able to catch the impossible flow in the second example above, but it can not handle all axes.

The XPath simulation proceeds by iterating through the steps in the location path being simulated. Each step describes an axis, a node test, and any number of predicates. However, due to the complexity of the predicates, we shall not examine them. The axes and node tests of the location steps

give rise to alternating operations on the set of node type found so far in the simulation. The axis operation finds—with the help of the input DTD—all node types that have the specified relation to nodes types in the result set of the last operation. For the `child` axis, this means finding the children of each node in the last set, and adding them to a new set. The node test operation will then filter the nodes resulting from the axis. For example, a name test will leave only the nodes with the specified name and which are of the principal node type.

In order to perform the axis operations, we will need a couple of *binary relations* describing relationships between the nodes on the input. The *possible child relation*, denoted PCR, describes what children each node type can have. This information relies exclusively on the content model for that node, so the relation can easily be constructed by inspecting the input DTD. Recall from Section 4.5.2 that we work with uniform content models, including—among other things—comments and processing-instructions in the models. This means that our task here becomes much simpler. The possible child relation can be described concisely as follows:

$$PCR \subseteq \Sigma \times \Sigma$$

$$PCR = \{(n, m) \mid n \in contentmodel(m)\}$$

This will take care of the `child` and `descendant` axes for us. Similarly, we must also construct a *possible parent relation*, denoted PPR. This relation is necessary solely because of the asymmetry in the child/parent relationships in the XPath data model. Attributes have a parent element, but they are not children of their parent.

$$PPR \subseteq \Sigma \times \Sigma$$

$$PPR = \{(n, m) \mid (m, n) \in PCR\} \cup \{(n, m) \mid m = (a, e) \in \mathcal{A} \times \mathcal{E} \wedge a \in attlist(n)\}$$

Thus the *upwards* axes, `parent` and `ancestor`, are taken care of. The rest of the axes are now easily handled, except `following`, `following-sibling`, `preceding`, and `preceding-sibling`. These four have a semantics not too easily handled. Although these axes could be handled more precisely, we shall—in light of their rare usage in practice—make rough approximations of their functionality.

The complete simulation process is described algebraically below. It is first defined how to convert an XPath location path to a series of alternating axis and node test operations. These operations are then defined individually for each axis and node test.

$$
\begin{aligned}
&S : \Phi \mapsto 2^{\Sigma}\\
&S^{step}_{a::t} : 2^{\Sigma} \mapsto 2^{\Sigma}\\
&S^{axis}_{a} : 2^{\Sigma} \mapsto 2^{\Sigma}\\
&S^{test\mathcal{E}}_{t} : 2^{\Sigma} \mapsto 2^{\Sigma}\\
&S^{test\mathcal{A}}_{t} : 2^{\Sigma} \mapsto 2^{\Sigma}
\end{aligned}
$$

$$
\begin{aligned}
&S(\epsilon) = \{\mathbf{root}\}\\
&S(a :: t) = S^{step}_{a::t}(\Sigma))\\
&S(P/a :: t) = S^{step}_{a::t}(S(P)))
\end{aligned}
$$

$$
S^{step}_{a::t}(\Delta) = \begin{cases} S^{test\mathcal{A}}_{t}(S^{axis}_{attribute}(\Delta)) & \text{if a} = \texttt{attribute}\\ S^{test\mathcal{E}}_{t}(S^{axis}_{a}(\Delta)) & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
&S^{axis}_{child}(\Delta) = \{n \in \Sigma \mid \exists m \in \Delta : (n, m) \in PCR\}\\
&S^{axis}_{descendant}(\Delta) = {S^{axis}_{child}}^{*}(\Delta)\\
&S^{axis}_{attribute}(\Delta) = \{(a, e) \in \mathcal{A} \times \mathcal{E} \mid e \in \Delta\}\\
&S^{axis}_{self}(\Delta) = \Delta\\
&S^{axis}_{descendant-or-self}(\Delta) = S^{axis}_{descendant}(\Delta) \cup S^{axis}_{self}(\Delta)\\
&S^{axis}_{parent}(\Delta) = \{n \in \Sigma \mid \exists m \in \Delta : (n, m) \in PPR\}\\
&S^{axis}_{ancestor}(\Delta) = {S^{axis}_{parent}}^{*}(\Delta)\\
&S^{axis}_{ancestor-or-self}(\Delta) = S^{axis}_{ancestor}(\Delta) \cup S^{axis}_{self}(\Delta)\\
&S^{axis}_{following-sibling}(\Delta) = S^{axis}_{preceding-sibling}(\Delta) = S^{axis}_{child}(S^{axis}_{parent}(\Delta - \mathcal{A} \times \mathcal{E}))\\
&S^{axis}_{following}(\Delta) = S^{axis}_{preceding}(\Delta) = \Sigma - (\mathcal{A} \times \mathcal{E})
\end{aligned}
$$

$$
\begin{aligned}
&S^{test\mathcal{E}}_{qname}(\Delta) = \Delta \cap \{n \in \mathcal{E} \mid name(n) = qname\}\\
&S^{test\mathcal{E}}_{*}(\Delta) = \Delta \cap \mathcal{E}\\
&S^{test\mathcal{A}}_{qname}(\Delta) = \Delta \cap \{n \in \mathcal{A} \times \mathcal{E} \mid name(n) = qname\}\\
&S^{test\mathcal{A}}_{*}(\Delta) = \Delta \cap \mathcal{A} \times \mathcal{E}\\
&S^{test\mathcal{E}}_{node()}(\Delta) = S^{test\mathcal{A}}_{node()}(\Delta) = \Delta\\
&S^{test\mathcal{E}}_{text()}(\Delta) = S^{test\mathcal{A}}_{text()}(\Delta) = \Delta \cap \{\mathbf{pcdata}\}\\
&S^{test\mathcal{E}}_{comment()}(\Delta) = S^{test\mathcal{A}}_{comment()}(\Delta) = \Delta \cap \{\mathbf{comment}\}\\
&S^{test\mathcal{E}}_{processing-instruction()}(\Delta) = S^{test\mathcal{A}}_{processing-instruction()}(\Delta) = \Delta \cap \{\mathbf{pi}\}
\end{aligned}
$$

In the above, ${S^{axis}_{child}}^{*}(\Delta)$ and ${S^{axis}_{parent}}^{*}(\Delta)$ refer to the *transitive closure* of the operations, i.e. continually applying the operation and adding the result to the previous set, until the set remains unchanged after an application. This can be done in a finite number of steps, since the operations never remove nodes from the set, and since there is a finite amount of nodes in Σ.

The reason for splitting the node test operation into $S^{test\mathcal{E}}_{t}$ and $S^{test\mathcal{A}}_{t}$ is that the node tests *`qname`* and `*` depend on the principal node type. And as can be seen in the above formalization, we approximate the `following` and `preceding` axes with simply $\Sigma - (\mathcal{A} \times \mathcal{E})$, at least taking advantage of the fact that neither axis can select attributes. Regarding the `following-sibling` and `preceding-sibling`, the following quote from the XPath 1.0 specification [25] can perhaps help to clarify the construction:

$S(\text{"}\texttt{parent::*/descendant-or-self::item/child::*/child::node()}\text{"}) =$
$S^{step}_{child::node()}(S^{step}_{child::*}(S^{step}_{descendant-or-self()::item}(S^{step}_{parent::*}(\Sigma)))) =$
$S^{test\mathcal{E}}_{node()}(S^{axis}_{child}(S^{test\mathcal{E}}_{*}(S^{axis}_{child}(S^{test\mathcal{E}}_{item}(S^{axis}_{descendant-or-self}(S^{test\mathcal{E}}_{*}(S^{axis}_{parent}(\Sigma)))))))) =$
$S^{test\mathcal{E}}_{node()}(S^{axis}_{child}(S^{test\mathcal{E}}_{*}(S^{axis}_{child}(S^{test\mathcal{E}}_{item}(S^{axis}_{descendant-or-self}(S^{test\mathcal{E}}_{*}(\{\mathbf{root}, \mathsf{news}, \mathsf{item}, \mathsf{headline}, \mathsf{text}, \mathsf{p}\}))))))) =$
$S^{test\mathcal{E}}_{node()}(S^{axis}_{child}(S^{test\mathcal{E}}_{*}(S^{axis}_{child}(S^{test\mathcal{E}}_{item}(S^{axis}_{descendant-or-self}(\{\mathsf{news}, \mathsf{item}, \mathsf{headline}, \mathsf{text}, \mathsf{p}\})))))) =$
$S^{test\mathcal{E}}_{node()}(S^{axis}_{child}(S^{test\mathcal{E}}_{*}(S^{axis}_{child}(S^{test\mathcal{E}}_{item}(\{\mathsf{news}, \mathsf{item}, \mathsf{headline}, \mathsf{text}, \mathsf{p}, \mathsf{item.category}, \mathsf{item.date}, \mathsf{item.time}, \mathbf{pcdata}\}))))) =$
$S^{test\mathcal{E}}_{node()}(S^{axis}_{child}(S^{test\mathcal{E}}_{*}(S^{axis}_{child}(\{\mathsf{item}\})))) =$
$S^{test\mathcal{E}}_{node()}(S^{axis}_{child}(S^{test\mathcal{E}}_{*}(\{\mathsf{headline}, \mathsf{text}\}))) =$
$S^{test\mathcal{E}}_{node()}(S^{axis}_{child}(\{\mathsf{headline}, \mathsf{text}\})) =$
$S^{test\mathcal{E}}_{node()}(\{\mathbf{pcdata}, \mathsf{p}\}) =$
$\{\mathbf{pcdata}, \mathsf{p}\}$

Figure 4.10: An example run of the location path simulation algorithm under the news DTD schema in Figure 2.2.

> "the following-sibling axis contains all the following siblings of the context node; if the context node is an attribute node or namespace node, the following-sibling axis is empty"

As mentioned earlier, the path simulation test now consists of simulating both the concatenated selection path and the prospective target match path. If the intersection of these two sets is empty, it is guaranteed that no edge flow to this target can exist, under the given context node. Otherwise, the intersection set describes possible edge flow, to be incorporated in the flow propagation algorith as described in Section 4.5.2.

An example run of the location path simulation on the news DTD of Figure 2.2 can be found in Figure 4.10.

4.5.4 Paths Automata Test

The path simulation test has some limitations that typically surface when a match path has more than one location step. If for example the concatenated selection is "a/b/c" and the match path "d/c". The path simulation test will determine c as possible selection of each path independently, and will thus mark the edge flow to be c. But obviously, c nodes from the above selection can never match on "d/c". They each require a different parent element.

In order to be able to catch these kinds of intricacies, we present the *paths automata flow propatagtion test*. The idea is to not only compare the final node types being produced by the paths, as the simulation test does, but also to consider their ancestors. Since the XSLT processing almost always recurses downwards, knowledge about the ancestors are often the distinguishing factor in template rule matching.

Ignoring the actual selection and match process, the selection and match paths for some candidate edge both simply express a number of constraints on nodes. In particular, this means that, after selection of some node-set by a selection path, each node in that node set will meet the constraints which the selection path expresses. If a node leads to instantiation of a template rule, this means that the node will further meet the constraints set by the match pattern.

While the predicates of a path can express all sorts of constraints, the actual path, consisting of the axes and node tests, can only constrain the node type and *ancestry* of a node. The ancestry is simply the node types and order of all ancestors of a node. So the text nodes in the news document of Figure 2.1 all have the ancestry **root** · news · item. These constraints on ancestries are much simpler to handle than the general XPath expressions of the predicates, and they usually take on the dominant role in controlling the processing of XSLT transformations. This fact is what we shall take advantage of.

The basic idea is—for each path—to produce a regular expression describing all possible *top-down paths* which the given selection or match path can accept, and then to compare these paths. A top-down path in some XML document is a path from the root to some node in the tree (possibly the root itself) only by "child or attribute steps". For example the path (**root**, news, item(1), headline) is a top-down path in the example news XML document of Figure 2.1. The "(1)" annotation should be read simply as "child number 1". Other examples of paths in the news example would be: (**root**, news, item(1)), (**root**, news, item(2), text, p(1), **pcdata**), and (**root**).

However, we are not working on document instances, so we need a more abstract way to express these paths: A *path expression* shall be a regular expression over a set of node types, that expresses a set of top-down paths in documents that use that set of node types. For example, the path expression (**root** · news · item · headline) expresses all top-down paths to headlines in our news documents. (**root** · news · item$^?$ · headline) expresses the same paths, as well as some paths that can never happen in documents conforming to the news DTD (Figure 2.2), namely (**root** · news · headline) paths, but we know headline can never be a child of a news element. Thus, these extra paths are "empty" under the news DTD.

Now, the idea behind the path expression construction is that a path such as `"/child::a/descendant::b"` shall produce a regular top-down expression such as: (**root** · a · $\mathcal{E}^*$ · b). In this simple case with only `child` and `descendant` axes, the construction is simple. The path is already in a top-down form, so the conversion is straight forward. However, when self-selection becomes involved with the `self` and `descendant-or-self` axes, the expressions become somewhat more complex as we shall see.

In order to build path expressions for the concatenated selection path and the match path, we shall simply "convert" each location step to a regular expression over Σ. This will be straight forward for the *downwards axes* such as `child` and `descendant`, but the *upwards axes* pose a serious problem. We can not directly describe the behavior of upwards location steps with top-down paths. The same can be said for the *sideways axes* such as `following` and `following-sibling`. Also, a relative path does not in itself describe top-down paths. A relative path may start from any context node in the input tree.

We shall tackle these issues firstly by approximating away any "non-downwards behavior". This means removing location steps from the left of the path until only downwards or `self` axis steps are left. Next, any relative path shall be prefixed with `"/descendant-or-self::node()/"`. This effectively makes the path top-down, i.e. it starts from the root, while still describing the same constraints on the ancestors of the nodes which the path accepts. Finally, we shall remove any predicates from the path as well. Predicates describe complex constraints on the accepted nodes, and these we shall not be able to express.

As an example of this path "preparation", the path `"parent::node()/child::item[@category]"` will be approximated with simply `"/descendant-or-self::node()/child::item"`. In this particular case, no information is in fact lost. The parent step expresses nothing but the fact that the item element must have a parent, which is already given, and according to the news DTD the category attribute specifies a default value for category attributes. Thus an item element will always have a category attribute, leaving the predicate always true. However, in general, information will be lost, but it is a sound approximation upholding the conservatism of the analysis.

Regular Path Expression Construction

We express the process of generating the path expressions by recursive definitions. The expressions make use of intersection operations in addition to the usual sequence, union, and Kleene * operations. ϵ denotes the empty string.
Formally, the path expression construction proceeds as follows:

$$R^{descend}(a) = \begin{cases} \mathcal{E}^* & \text{if a} = \texttt{descendant} \vee \texttt{descendant-or-self} \\ \epsilon & \text{otherwise} \end{cases}$$

$$\widehat{R^{step}_{att}}(t) = \begin{cases} (e_1, t)|\ldots|(e_n, t) \;\; \forall e_i \in \mathcal{E} & \text{if t}=qname \wedge \exists(e_i, t) \in \mathcal{A} \\ \mathcal{A} & \text{if t}=\texttt{node()} \vee \text{t}=\texttt{*} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\widehat{R^{step}_{self}}(t) = \begin{cases} t & \text{if t=}qname \\ \mathcal{E} & \text{if t=}\texttt{*} \\ \Sigma & \text{if t=}\texttt{node()} \\ \textbf{pcdata} & \text{if t=}\texttt{text()} \\ \textbf{comment} & \text{if t=}\texttt{comment()} \\ \textbf{pi} & \text{if t=}\texttt{processing-instruction()} \end{cases}$$

$$\widehat{R^{step}_{child}}(t) = \begin{cases} \mathcal{E} \cup \{\textbf{pcdata}, \textbf{pi}, \textbf{comment}\} & \text{if t=}\texttt{node()} \\ \widehat{R^{step}_{self}}(t) & \text{otherwise} \end{cases}$$

$$R^{step}_{a}(t) = \begin{cases} \widehat{R^{step}_{att}}(t) & \text{if a=}\texttt{attribute} \\ \widehat{R^{step}_{self}}(t) & \text{if a=}\texttt{self} \vee \text{a=}\texttt{descendant-or-self} \\ \widehat{R^{step}_{child}}(t) & \text{if a=}\texttt{child} \vee \text{a=}\texttt{descendant} \end{cases}$$

$$R(\epsilon) = \textbf{root}$$

$$R(P/a :: t) = \begin{cases} R(P/\texttt{descendant::t}) \mid R(P/\texttt{self::t}) & \text{if a=}\texttt{descendant-or-self} \\ R(P) \cap (\Sigma^* R^{step}_{a}(t)) & \text{if a=}\texttt{self} \\ R(P) R^{descend}(a) R^{step}_{a}(t) & \text{if a=}\texttt{child} \vee \text{a=}\texttt{descendant} \vee \\ & \quad \text{a=}\texttt{attribute} \end{cases}$$

Note that the name step always works only on the principal node type.

Let us examine how the algorithm constructs a path expression for the small example "/child::a/descendant::b", mentioned earlier:

$$\begin{aligned} & R(\texttt{/child::a/descendant::b}) = \\ & R(\epsilon) \cdot R^{descend}(child) \cdot R^{step}_{child}(a) \cdot R^{descend}(descendant) \cdot R^{step}_{descendant}(b) = \\ & \textbf{root} \cdot \epsilon \cdot \widehat{R^{step}_{child}}(a) \cdot \mathcal{E}^* \cdot \widehat{R^{step}_{descendant}}(b) = \\ & \textbf{root} \cdot \widehat{R^{step}_{self}}(a) \cdot \mathcal{E}^* \cdot \widehat{R^{step}_{self}}(b) = \\ & \textbf{root} \cdot \mathsf{a} \cdot \mathcal{E}^* \cdot \mathsf{b} \end{aligned}$$

As we can see, the expression construction performs as expected on this example. In the calculations above, $\cdot$ denotes concatenation to make the expressions more readable. A more complex example run of the path expression construction can be found in Figure 4.11.

After construction of a regular expression, we need to convert the expressions to *deterministic finite state automata* in order to be able to intersect with other regular path expressions and the DTD. Such an automaton construction shall be denoted $FA(exp)$ for some regular path expression *exp*.

An important property of these resulting automata is that the *accept states* of the automata describe all the node types which occur as last step in a top-down path: More precisely, the incoming edges of the accept states

$\mathrm{R}(\texttt{/descendant-or-self::node()/child::*/self::item/child::text/child::*}) =$
$\mathrm{R}(\texttt{/descendant-or-self::node()/child::*/self::item/child::text})R^{descend}(\texttt{child})R^{step}_{child}(\texttt{*})$
$=$
$\mathrm{R}(\texttt{/descendant-or-self::node()/child::*/self::item/child::text})\ \epsilon\ \widehat{R^{step}_{child}}(\texttt{*}) =$
$\mathrm{R}(\texttt{/descendant-or-self::node()/child::*/self::item/child::text})\ \widehat{R^{step}_{self}}(\texttt{*}) =$
$\mathrm{R}(\texttt{/descendant-or-self::node()/child::*/self::item/child::text})\ \mathcal{E} =$
$\mathrm{R}(\texttt{/descendant-or-self::node()/child::*/self::item})\ R^{descend}(\texttt{child})\ R^{step}_{child}(\texttt{text})\ \mathcal{E} =$
$\mathrm{R}(\texttt{/descendant-or-self::node()/child::*/self::item})\ \texttt{text}\ \mathcal{E} =$
$(\mathrm{R}(\texttt{/descendant-or-self::node()/child::*}) \cap (\Sigma^* R^{step}_{self}(\texttt{item})))\ \texttt{text}\ \mathcal{E} =$
$(\mathrm{R}(\texttt{/descendant-or-self::node()/child::*}) \cap (\Sigma^*\ \texttt{item}))\ \texttt{text}\ \mathcal{E} =$
$((\mathrm{R}(\texttt{/descendant-or-self::node()})\ R^{descend}(\texttt{child})R^{step}_{child}(\texttt{*})) \cap(\Sigma^*\ \texttt{item}))\ \texttt{text}\ \mathcal{E} =$
$((\mathrm{R}(\texttt{/descendant-or-self::node()})\ \epsilon\mathcal{E}) \cap(\Sigma^*\ \texttt{item}))\ \texttt{text}\ \mathcal{E} =$
$(((\mathrm{R}(\texttt{/descendant::node()})\ |\ \mathrm{R}(\texttt{/self::node()}))\ \mathcal{E}) \cap(\Sigma^*\ \texttt{item}))\ \texttt{text}\ \mathcal{E} =$
$((((R(\epsilon)R^{descend}(\texttt{descendant})R^{step}_{descendant}(\texttt{node()}))\ |\ (R(\epsilon)\cap(\Sigma^* R^{step}_{self}(\texttt{node()}))))\ \mathcal{E}) \cap(\Sigma^*\ \texttt{item}))$
$\texttt{text}\ \mathcal{E} =$
$((((\mathbf{root}\mathcal{E}^*(\mathcal{E} \cup \{\mathbf{pcdata}, \mathbf{pi}, \mathbf{comment}\}))\ |\ (\mathbf{root} \cap (\Sigma^*\Sigma)))\ \mathcal{E}) \cap(\Sigma^*\ \texttt{item}))\ \texttt{text}\ \mathcal{E} =$

Which is equivalent to:
$(\mathbf{root} \cdot \mathcal{E}^* \cdot (\mathcal{E} \cup \{\mathbf{pcdata}, \mathbf{pi}, \mathbf{comment}\}) \cdot \texttt{item} \cdot \texttt{text} \cdot \mathcal{E})\ |\ (\mathbf{root} \cdot \texttt{item} \cdot \texttt{text} \cdot \mathcal{E})$

And intersected with news DTD of Figure 2.2, it becomes simply:

$\mathbf{root} \cdot \texttt{news} \cdot \texttt{item} \cdot \texttt{text} \cdot \texttt{p}$

Figure 4.11: A complex example run of the regular path expression construction algorithm.

are labeled with all the possible last step node types. These node types are the node types which can be selected by a selection path, or matched by a match path. We shall make use of this fact for extracting the resulting edge flow of this flow propagation test.

The DTD Paths Automaton

In order to be able to intersect the paths automata of the selection and match paths with the input DTD, we need to be able to construct a finite automaton for this DTD as well. This leads us to a *DTD paths automaton*. The construction is rather simple. We do not need to concern ourselves with the order and cardinality of siblings expressed in the content models of the DTD. We only need the *possible child relation* constructed in Section 4.5.3, which is derived from the content models.

The construction process proceeds simplest by constructing a deterministic finite automaton directly. Given $D \in DTD$, we shall construct FA_D as follows:

- Each node type in Σ corresponds to a state in the automaton, which we shall here denote $state(n)$ for some node type $n \in \Sigma$.
- For each possible child m of a node n, an edge from $state(n)$ to $state(m)$ with label m is constructed.
- The initial node is denoted initial, and it has a single edge to $state(\mathbf{root})$.

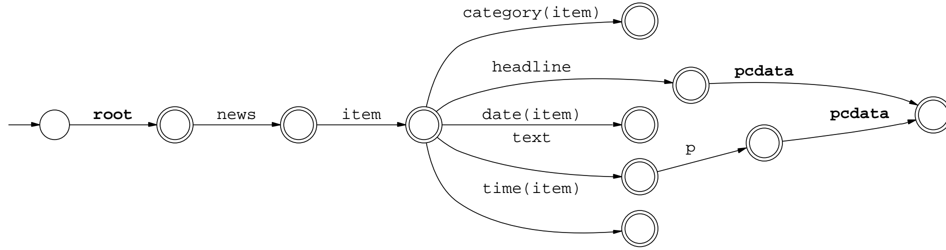


Figure 4.12: The paths automaton for the news DTD schema from Figure 2.2. The otherwise ubiquitous edges and states for comments and processing-instructions—which are allowed almost everywhere—have been removed for a better overview.

- All states are accept states, except the initial state initial. This means that empty paths are not allowed.

Note that, in the resulting automaton, all edges going into a state $state(n)$ are labeled with the node type n. The DTD automaton can be described more formally, using the standard notation of $FA_D = (Q, \Sigma_D, q_0, \delta, A)$, defining the set of states, input alphabet, initial state, transition function, and accept states of the automaton respectively:

$$
\begin{array}{|l|}
\hline
\Sigma_D = \mathcal{E}_D \cup (\mathcal{A}_D \times \mathcal{E}_D) \cup \{\mathbf{root}, \mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\} \\
Q = \Sigma_D \cup \{\mathsf{initial}\} \\
q_0 = \mathsf{initial} \\
A = \Sigma_D \\
\forall n, m \in \Sigma_D : (n, m) \in PCR_D \Rightarrow \delta(n, m) = m \\
\delta(\mathsf{initial}, \mathbf{root}) = \mathbf{root} \\
\hline
\end{array}
$$

Here, PCR_D refers to the possible child relation constructed from D. The construction of FA_D can easily be implemented in time proportional to the automaton constructed with the right data structures. The paths automaton for the input DTD of our running news example can be found in Figure 4.12.

This construction is very similar to the *DTD-Graphs* of Dong and Bailey in [32], except for the fact that we here use all the node types of the XPath data model, as expressed in our extended content models.

The Flow Propagation Test

Finally, the paths automata flow propagation test shall consist of intersecting the automata for the selection path, the match path of the prospective target, and the DTD. If the resulting automaton is not empty, we can—as explained earlier—extract the edge flow by examining incoming edges of the accept states of the intersected automaton. These are the node types which

can both be selected by the selection path, matched by the match path, and which are meaningful under the input DTD. More precisely:

Given a candidate flow graph edge $(n_s,\, S,\, n_t) \subseteq N \times \mathcal{P}_s \times N$, and a context node $c \in \mathcal{C}(n_s)$ in the source context set, let:

$$FA_\cap = FA(CONCAT(match(n_s),\, c,\, S)) \cap FA(match(n_t)) \cap FA_{D_{in}}$$
$$FPT_{aut} = \{n \in \Sigma \mid \exists a \in A_{FA_\cap}, s \in Q_{FA_\cap} : \delta_{FA_\cap}(s, n) = a\}$$

FPT_{aut} defines the edge flow of the path automata flow propagation test, while $Q_{FA_\cap}$, $A_{FA_\cap}$, and $\delta_{FA_\cap}$ refer to the states, accept states, and transition function of $FA_\cap$.

Note that the regular path expression construction itself is complete in the sense that no information is lost. It is in the path preparation where information is potentially lost, and this is the source of inaccuracy for this flow propagation test.

4.5.5 The Priority Override Filter

As described Section 4.5.2 we need to find an algorithm that can determine when flow on a higher priority edge overrides flow on a lower priority edge. This algorithm will need exact knowledge of the flow along the higher priority edge in relation to the lower priority edge: Does node type n *always* flow to the higher priority edge when given the choice between the two given edges? If it does not, some nodes of the given node type can still flow into the low priority template rule. Since we have no way of expressing that only a specific fraction of n flows along an edge, this is not good enough. We have chosen only to work with whole node types.

The conservatism in the approximated context sets means that we cannot use them in this priority filtering: They contain no knowledge about what can with certainty flow along some edge, only knowledge of what *might* flow along an edge. We must instead examine the two flow graph nodes in relation to each other. There are three properties which influence whether a template rule matches some node: (1) The match pattern, (2) the priority, and (3) the mode. When filtering flow by priority, the targets of the two edges will have the same mode, and we are only interested in pairs of edges where one has higher priority than the other. Now, all that differentiates the two targets from each other is the match paths. Let us formulate the priority filtering problem more precisely:

> *Given edges e_1 and e_2 where $priority(e_1) > priority(e_2)$, and $n \in \mathcal{F}(e_1) \cap \mathcal{F}(e_2)$. We must guarantee that whenever a node of type n can match $match(e_2)$, it can also match $match(e_1)$.*

If this guarantee can be made, then we know that flow will always prefer e_1 over e_2, and thus nodes of type n can never flow along e_2.

So, we need some kind of inclusion test between two match paths. For this, we can use the paths automata of Section 4.5.4. If we, for each of the two edges, could construct an automaton describing possible top-down paths that the target match path can match under the given concatenated selection path, it would be a simple matter of checking inclusion of the automaton for e_2 in the automaton for e_1. However, there are two problems: (1) The paths automaton algorithm can not handle all axes, and (2) predicates on the location path degrades the precision of the automaton. Only a fragment of the XPath location paths are allowed in the match expressions because they are XPath patterns. These patterns are restricted to the `child`, `attribute`, and `descendant-or-self` axes, which means that our paths automaton algorithm can construct an automaton for any match path. We also wish to take the concatenated selection path into account, but this will not always be possible. At least not the full concatenated selection. Since it is a sound and conservative approximation to leave out the concatenated selection, we shall simply include as much of it as possible by stripping away non-downwards axes as described in Section 4.5.4.

Now, the problem with predicates are that they can filter out any number of nodes in the edge flow. In the extreme case where the target match path contains a `[false]` predicate, the target template rule can never match on anything, and all flow will go somewhere else. For example to a built-in template rule. This example serves well to illustrate that, without some kind of analysis on the predicates, we can trust neither the edge flow, nor the paths automaton. Note however that since the predicates are *filters*, they can only remove nodes. For a match path, this means that a predicate can only shrink the set of nodes that the path can match. Thus, predicates on the match path of the lower priority edge are unimportant. However, predicates on the match path of the higher priority edge can not be ignored.

Finally, incorporating the input DTD in the priority override shall provide some additional precision. For example, if we are examining two edges whose target match paths are "`*/b`" and "`a/b`", the second match path obviously expresses a subset of the paths which the first expresses. The opposite does not hold in general, but if the input DTD says that only `a` elements are allowed as parents of `b` elements, then the two match paths are suddenly equal. So we shall intersect with the paths DTD from Section 4.5.4 as well. To conclude, the priority override test will do the following:

> *Given edges e_1 and e_2 with the same origin, and under the same context node: Let S be the non-downwards-axis stripped concatenated selection path. If $priority(e_1) > priority(e_2) \land \exists n \in \mathcal{F}(e_1) \cap \mathcal{F}(e_2) \land e_1$ contains no predicates $\land FA(match(e_2)) \cap$*

$FA(S) \cap \Sigma^* n \cap FA_{D_{in}} \subseteq FA(match(e_1))$, then n is removed from $\mathcal{F}(e_2)$.

Essentially what the test is doing is—given e_1, e_2, and n as above—to: (1) Find all the top-down paths ending with the node type n, which $match(e_2)$ accepts: $FA(match(e_2)) \cap \Sigma^* n$. (2) Narrow these down to the paths which are meaningful in D_{in}: Intersect with $FA_{D_{in}}$. (3) Check whether all these paths are accepted by the higher priority match path by doing an inclusion test on $FA(match(e_2))$.

Recall that for this to be a sound priority override filter, we must "guarantee that whenever a node of type n can match $match(e_2)$, it can also match $match(e_1)$.". We shall argue as follows: Because e_1 contains no predicates, it exclusively describes constraints on the ancestries of nodes that it accepts, in addition to the node type itself. Since predicates filter out nodes by imposing additional contraints, a path *with* predicates can always accept only a subset of the nodes that the same path *without* predicates can. Thus we can safely ignore the predicates in $match(e_2)$, and assume that it has none: They only make our problem harder. Now, the property $FA(match(e_2)) \cap \Sigma^* n \cap FA_{D_{in}} \subseteq FA(match(e_1))$ implies that all nodes of type n which $match(e_2)$ accepts will have ancestries which $match(e_1)$ accepts. In other words: Every node of type n accepted by $match(e_2)$ is also accepted by $match(e_1)$. Since e_1 has the higher priority, we know with certainty that no nodes of type n can ever flow to the target of e_2.

Let us examine a couple of examples to illustrate the process. Examining two edges, each from the same selection, and each annotated with the edge flow $\{\mathsf{d}\}$ for the given context node:

$$\begin{array}{ll} priority(e_1) = 1 & priority(e_2) = -1 \\ match(e_1) = "a/d" & match(e_2) = "b/d" \\ \mathcal{F}(e_1) = \{\mathsf{d}\} & \mathcal{F}(e_2) = \{\mathsf{d}\} \end{array}$$

This leads to the path automata: $R("a/d") = (\mathbf{root} \cdot \mathcal{E}^* \cdot \mathsf{a} \cdot \mathsf{d})$, and $R("b/d") = (\mathbf{root} \cdot \mathcal{E}^* \cdot \mathsf{b} \cdot \mathsf{d})$. But $R("b/d") \cap \Sigma^* \mathsf{d} = R("b/d") \not\subseteq R("a/d")$

Another example is:

$$\begin{array}{ll} priority(e_1) = 1 & priority(e_2) = -1 \\ match(e_1) = "c/d" & match(e_2) = "b/c/*" \\ \mathcal{F}(e_1) = \{\mathsf{d}\} & \mathcal{F}(e_2) = \{\mathsf{d}, ...\} \end{array}$$

The path automata become: $R("c/d") = (\mathbf{root} \cdot \mathcal{E}^* \cdot \mathsf{c} \cdot \mathsf{d})$, and $R("b/c/*") = (\mathbf{root} \cdot \mathcal{E}^* \cdot \mathsf{b} \cdot \mathsf{c} \cdot \mathcal{E})$. And $R("b/c/*") \cap \Sigma^* \mathsf{d} = (\mathbf{root} \cdot \mathcal{E}^* \cdot \mathsf{b} \cdot \mathsf{c} \cdot \mathsf{d}) \subseteq R("c/d")$. Thus we can remove d from $\mathcal{F}(e_2)$.

4.5.6 The Result

Summarizing, we have now described our XSLT flow graphs, and how they are constructed by propagating flow around in the graph, examining prospective edges with the propagation tests and the priority filter. An example of the outcome of our analysis has already been introduced in Figure 4.9. We shall now move on to how we can make use of the constructed flow graph.

4.6 Parameter Analysis

Parameters are a convenient and regularly used construct, but they can become serious obstacles for our analysis, if used in certain ways. To adress the most obvious, top-level parameters—which can be passed to the XSLT processor at each transformation run—can potentially contain any values at runtime. They can even be of any type. Although each parameter must define a default value, which indicates what the expected input to this parameter is, there are in fact no constraints on that input. This means that if the part of the transformation we are analyzing depends on such a top-level parameter, we can do nothing but to give up. Alternatively, we can report the validity error, and continue analysis with the default value provided. But we can in such a case never fully guarantee output validity.

But when is the analysis dependent on top-level parameters? References to parameters in `value-of` instructions are of little importance. We do not examine text in the output much farther than if text is allowed at some point or not. The output DTD is unable to describe further constraints anyway. A parameter reference in an `apply-templates` selection path definitely matters for our analysis. But this particular construct is very hard to analyze: The possible values of parameters depend on the flow analysis, and conversely, the flow analysis depends on the `apply-templates` selection paths. As such, this construct bites itself in the tail. We must again give up in such a case.

However, single parameter references in the select expressions of `copy-of` instructions are another matter. From the simplification phase of Section 4.4, we have reduced `copy-of` instructions to `apply-templates` instructions, except with these singleton parameter references such as "$\$p$". In the case of p being a local parameter, we must examine the flow graph to find possible values for insertion. When p is a top-level parameter, we again know nothing, and must report an error.

Note also that top-level parameters can flow into the local parameters, and thus create errors on local parameter references as well.

Analyzing the flow graph for local parameter values is rather simple. Due to the flow graph being an approximation, the possible parameter values will of course be an approximation as well.

For every parameter reference "$$p$" in `copy-of` instructions, and for every incoming edge to the containing flow graph node:

- If there is no `with-param` instruction for p at the source of the edge, then the default value specified in the `param` instruction is a possible value.
- If there is a `with-param` instruction for p at the source of the edge, then the value specified in the `with-param` instruction is a possible value.

Note that, if the `with-param` is further dependent on something, this must be resolved first. Cycles, we cannot analyze.

This analysis is rather rough, and it operates independent of the context sets and other specific flow information. However, this is enough to catch for example simple parameter passing to named templates, as they will have only a single incoming edge in the flow graph.

When the selections in `copy-of` instructions have been resolved, we wish to reduce the instructions to the other constructs of reduced XSLT, just as we did in the simplification phase. With the parameter analysis in hand we shall do the following:

- If the referenced parameter can have more than one value, we construct a `choose` instruction with a branch for each possible value of the parameter.
- If the parameter is of the *result tree fragment* type, we simply insert the fragment in place of the `copy-of` instruction. The result tree fragment is already simplified.
- If the parameter is of the *node-set* type, we convert the `copy-of` instruction to an `apply-templates` instruction and some template rules, just as is done in the simplification phase.
- If the parameter is of string or number type, we convert to a `value-of` instruction.

As it turns out, `copy-of` instructions are mostly used in conjunction with multiple input documents. Since we can not handle these multiple inputs, our parameter analysis is in fact rarely useful in practice. Nevertheless, the principles should be useful if one were to extend our analysis with these multiple input schemas.

4.7 The Summary Graph Analysis

The summary graph abstraction comes originally from "Static validation of dynamically generated HTML" [13] by Brabrand, Møller, and Schwartzbach. It has later been seen in various alternative forms in JWIG [21] and Xact [51]. We shall present here yet another alternative definition of summary graphs, suited to our needs. Our alterations shall require modifications to the summary graph inclusion algorithm, but the changes to the graphs and the inclusion analysis are not fundamental. We shall use the summary graph definition of JWIG as a starting point.

Recall that a summary graph is essentially a number of well-balanced XML fragments called *templates*. Each template may, in addition to XML content, contain *gaps*, which are named insertion points. The *edges* of the summary graph describe what content may be inserted at which gaps.

First off, the version of summary graphs we will use in this analysis does not contain a *gap presence* map, which in JWIG is used to detect various programming errors concerning the use of gaps. Since we will be constructing the graph ourselves, this is not needed. The *code gaps* of JWIG have no relevance in our context either.

Next, we need to handle attributes more generally. The problem is that attributes can be constructed with `attribute` instructions in XSLT. These `attribute` instructions are conceptually disconnected from their parent, and can be involved in arbitrarily complex template rule recursions, before getting instantiated. This complexity is not directly supported in the summary graph model of JWIG, and neither is it in any of the other formulations of summary graphs which have been published. We shall call these disconnected attributes for *floating attributes*. In our summary graph model, all attributes are modeled as floating attributes, and they occur in the summary graph node templates along with the other ordered content such as elements, text, and gaps. This will describe such properties as whether an attribute always is present on some element. Note that this abstraction subsumes the *attribute gaps* of the JWIG summary graphs, since these are easily modeled with a floating attribute and the normal gap structure. However, JWIG models string values—and thereby also attribute values—much more precisely than we will need, through general regular expressions. The DTDs we work with can only express very loose constraints on string values.

Note that there is now information about the ordering of attributes in the summary graph. This extra information might seem superfluous considering the unorderedness of attributes in XML, and it is to some degree, but taking a look at the semantics of the `attribute` instructions in XSLT1.0, we see that it is not an error if more than one `attribute` instruction constructs the same attribute on some element. The value of the attribute will then simply be the value constructed in the last of the `attribute` instructions.

$$
\begin{aligned}
\mathit{xmltemplate} &::= \mathit{content}^* \\
\mathit{content} &::= \mathit{element} \mid \mathit{attribute} \mid \mathit{gap} \\
\mathit{element} &::= \texttt{<element name="}\mathit{name}\texttt{" >}\mathit{xmltemplate}\texttt{</element>} \\
\mathit{attribute} &::= \texttt{<attribute name="}\mathit{name}\texttt{" >}\mathit{gap}\texttt{</attribute>} \\
\mathit{gap} &::= \texttt{<sg:}\mathit{name}\texttt{/>}
\end{aligned}
$$

Figure 4.13: The grammar for XML templates in the summary graphs. *name* refers to a legal element name under the XML specification.

For the summary graph model, this means that the order of the floating attributes does in fact make a difference. We shall not make perfect use of this information though, as we shall see in Section 4.7.2.

The summary graph templates of our model can be derived from the *xmltemplate* non-terminal in the grammar depicted in Figure 4.13. These summary graph templates, which we shall denote T, form the *nodes* of the summary graph, and we shall use the two terms: templates and nodes interchangeably. Let further E_t, and E_s be the *template edges* and *string edges* of the summary graph respectively. The template edges describe insertion of XML fragments, while the string edges denote string insertions. A summary graph shall now be defined as:

$$SG = (\widehat{T}, E_t, E_s)$$

Where:

$\widehat{T} \subseteq T$ is the *root nodes* of the summary graph.
$E_t \subseteq T \times G \times T$ represents the *template edges*, while
$E_s : T \times G \mapsto 2^{\mathbb{S}} \cup \{ANY\}$ is the *string edge map* of the graph.

Here, ANY refers to the set of all strings, which can be expressed for example as a finite state automaton.

The summary graph expresses a set of XML documents, and in the spirit of the JWIG summary graph terminology, we shall call one such XML document, which the summary graph expresses, for an *unfolding*. The *unfolding relation*, which describes all the possible unfolding of a summary graph, is for our summary graphs essentially the same as for the JWIG summary graphs, except for the floating attributes. We shall not go into too much detail in this, but instead describe the unfolding relation informally as follows:

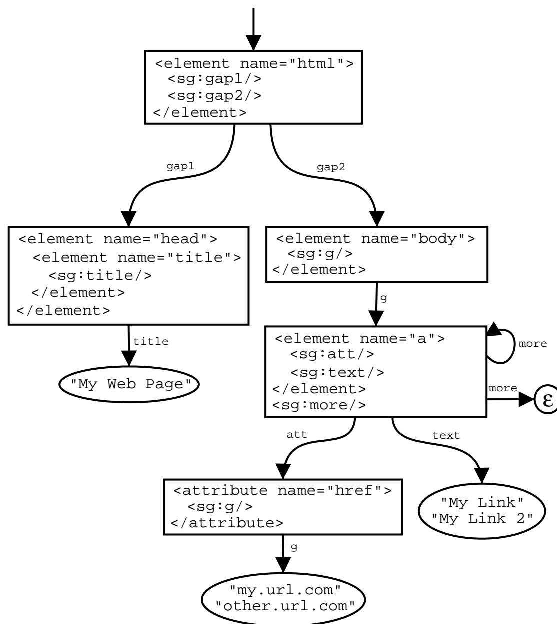


Figure 4.14: Example summary graph.

```
<html>
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <a href="my.url.com">
      My Link
    </a>
    <a href="other.url.com">
      My Link 2
    </a>
  </body>
</html>
```

Figure 4.15: One possible unfolding of the example summary graph in Figure 4.14.

> *An unfolding of a summary graph is produced by starting at one of the root templates, and then replacing all gaps in the template with either: (1) an unfolding of the summary graph template pointed to by one of the template edges, or (2) one of the strings described by a string edge. Finally, each* `element` *is replaced by an element of the given name, and each* `attribute` *is turned into an attribute with the given name on its—now unfolded—parent element.*

An illustration of a summary graph can be found in Figure 4.14, while Figure 4.15 describes one of the possible unfoldings for that summary graph.

4.7.1 Summary Graph Construction

As discussed in Section 4.5.2, we will construct a summary graph node for each flow graph node and node type pair. That is, each flow graph node will correspond to a summary graph node for each of the node types in its context set. This extra dimension allows a much more precise expression of the flow of construction of an XSLT transformation.

Each flow graph node originates from a certain template rule, and this template rule will form the base for constructing the summary graph templates. Recall that the templates in reduced XSLT consist exclusively of XSLT instructions (See the grammar in Figure 4.5). Also note that the `copy-of` instructions have been handled as far as possible at this point, so none of those are left.

The roots of the output summary graph we are about to construct is are the summary graph templates constructed from the roots of the flow graph. In other words, the root templates correspond to the roots of the flow graph. Now, given a flow graph node and a context node from its context set, we shall convert each instruction in the template rule corresponding to the flow graph node as follows:

- `apply-templates`: These instructions take care of all recursion in reduced XSLT. Given a context node, the instruction selects a node-set from the input, and instantiates a template rule for each of the nodes in the node-set, with that node as context node. All the instantiations are then concatenated in document order of their context nodes, and inserted in place of the `apply-templates` instruction.
  In order to model this in the summary graph, we must first model the order and cardinality of the nodes in the selected node-set in document order, and then model which templates are possible for instantiation for the given node. The summary graph fragment modeling the selected node-set shall be called a *selection fragment*, while the connection to possible target rules shall be constructed through *instantiation gaps and edges*. This construction shall be described in detail later in the section.

- `choose`: This instruction essentially describes a number of possible templates to be instantiated. After simplification, each of these templates are a single `apply-templates` instruction, so we shall simply construct summary graph fragments describing each of the branches, and then make a template edge to each. This expresses that any of the branches are possible.

- `element` or `attribute`: The element and attribute constructions can be converted almost directly to the summary graph template. However, the name attribute does not always specify a constant string name as the summary graph templates require. The `"xslv:unknownString()"` has already been sorted out, since they can never be guaranteed to conform to the output DTD. Now, all that is left to handle is the `"{local-name()}"` construct, which amount to simply inserting the name of the context node of our summary graph template. If the context node is a node which has no name, i.e. **root**, **comment**, or **pcdata**, this is an error, and in the case of a processing instruction,

the name can be anything, and we have to signal an error too. Finally, we must recurse on the content of the instruction.

- `value-of`: The text insertion of the `value-of` instruction is replaced by a gap, and a string edge expressing the value in the select attribute is inserted. If the selection is a *name*, the string edge maps to that name. If the selection is `"xslv:unknownString()"`, the edge shall point to *ANY*.

With the above conversion, we now have a small piece of summary graph for each flow graph node and possible context node pair. We shall call these pieces for *instantiation fragments*. Just as the execution of the transformation would result in alternating selections and template instantiations, our summary graph will basically be alternating between selection fragments and instantiation fragments. The instantiation fragments express the possible content resulting from a particular template instantiation, while the selection fragments model how the selection node-sets are structured.

Content Model Fragments

The selection node-sets very often select children of the context node. This means that the node-set will mirror the content model of the context node. Similarly, if children of children are selected, the node-set will mirror several content models inserted into each other. So we need to be able to construct summary graph fragments mirroring these content models. These we shall call *content model fragments*. The construction of a content model fragment proceeds very similarly to the DTD to summary graph conversion described in Xact [51], except that here we work with the extended content models of Section 4.5.2.

Each regular expression operator in the content model will be converted to summary graph constructs as follows:

- `(..,..,..)`: Sequence results in a single summary graph template with a gap for each sub-expression of the sequence. Each gap shall have a single template edge pointing to the summary graph fragment corresponding to the respective sub-expression.

- `(..|..|..)`: Union results instead in a template with only a single gap, and with a template edge from this gap to each of the fragments corresponding to a sub-expression.

- `(..)+`: One or more is modeled by summary graph node with a small loop. The template will be: `"<sg:g/><sg:more/>"`. The gap `g` shall have a template edge to the fragment of the sub-expression, while the `more` gap shall have an edge pointing to the template itself. This

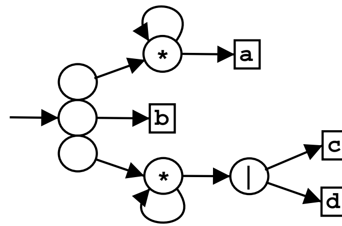


Figure 4.16: An illustration of the content model fragment for the content model: $(a*, b, (c|d)*)$. The boxes represent instantiation gaps, while the circles describe the internal structure of the regular expression.

describes exactly one or more insertions of what the `g` template edge points to.

- `(..)?`: Optional is simply handled by inserting a template edge to the fragment of the sub-expression, and an edge to an empty template.
- `(..)*`: Kleene * is modeled as the composition of optional and one or more: `((..)+)?`.
- *name*: An element name is represented simply by a gap with a name that uniquely identifies this gap as an instantiation gap for the given element type.
- *node type*: The node types in $\{\mathbf{pcdata}, \mathbf{comment}, \mathbf{pi}\}$, introduced in the extended content models, produce uniquely named instantiation gaps the same as the element types.

After this construction, the content model fragment essentially expresses the same regular expression as the content model, but over instantiation gaps instead of node types. The idea is then to plug the possible template instantiations, or further content model fragments, into the instantiation gaps. An illustration of a content model fragment can be found in Figure 4.16. The example is taken from an actual content model of the `layout-master-set` element of the XSL Formatting Objects language.

In addition to the child elements handled above, our content model fragments must also support selection of attributes. Thus, our content model fragments shall be rooted at a summary graph template of the form: `"<sg:attributes/><sg:children/>"`. The `attributes` gap shall have a template edge pointing to a fragment expressing attribute selection, while the `children` gap has an edge pointing to the fragment constructed above from the element content model.

The trouble with the attribute part of the content model fragment is that attributes are unordered. According to the XSLT specification, the

order of attributes in the document order is implemementation specific. In practice, this typically means the order that the attributes occurred in the XML document, but relying on this would not be sound. We also can not feasibly express a single occurrence of each attribute in arbitrary order, in the summary graph. Thus, we must in general approximate the attributes with arbitrary order and cardinality.

However, there are two cases which can save us: When only one attribute is selected, then the order of the attribute gaps is inconsequential. Only the selected attribute will produce an intantiation edge. The other case is when only non-element content is produced by the attribute selections. A DTD is unable to contrain the order of anything but elements. Thus, if no possible template instantiation, following the selection of an attribute, can produce elements in the output, the order of the attribute instantiation gaps is again inconsequential.

One final issue to handle is the requiredness of the attributes in the input. Given that we did not have to approximate the attribute selection with the worst-case fragment, we must add an edge to an empty template when the given attribute does not always occur. This can only happen if the attribute is declared `IMPLIED` in the input DTD. Otherwise, the attribute is either declared `REQUIRED`, or a default value is specified.

Selection Fragments

We must now make use of the content model fragments, and construct the selection fragments, which model the order and cardinality of the document order in selection node-sets.

Using the data mining on select expressions in Figure 4.3 as a starting point, we shall construct selection fragments for the most used selection expressions. Note that we can ignore union expressions for now, as they have been pulled apart in the analysis. We shall take a look at the implications of this further down. It is very important that we handle the constructs created in our simplification phase, as these constructs will be abundant in reduced XSLT. This includes for example `"self::node()"`, which is used, among other things, for named template calling and `if`/`choose` de-nesting.

Each of the common selections shall result in the following selection fragments:

- `child::`*test*: A single child step shall result in a single content model fragment for the context node, plugging instantiation gaps according to the edge flow.

- `child::`$test_1$/`child::`$test_2$/`child::`$test_3$: A series of content model fragments concatenated. Each element matching the node test in a step, shall have its relevant content model inserted in the proper instantiation gaps of the previous step.

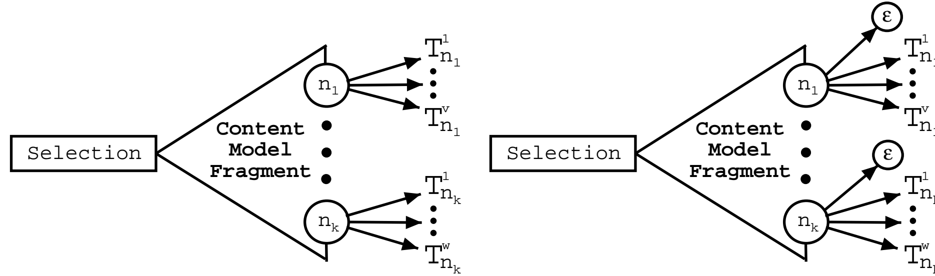


Figure 4.17: Selection fragment for: "`child::`*test*" and "`child::`*test*[...]".

- `..`: This selection will only ever contain a single node, namely the parent of the context node. Thus we simply add an edge to each of the possible targets. I.e. no instantiation gaps need to be constructed.

- `/`: This selection will only ever contain the root node. So we again just add an edge to each of the possible targets.

- `self::node()`: This selection is used extensively in our simplification constructs, so we must handle it. The selection will always only contain the context node. Thus—like with the parent and root selections—we simply add an edge to each of the possible targets.

- If none of the above, we construct the *worst-case construct* representing *any order and cardinality*. This corresponds to the content model fragment of Σ^* (treating attributes the same as the rest of the node types). Or altenatively $(n_1|n_2|\cdots|n_k|$, where n_i are all the node types which have possible targets for this selection and context node.

After construction of the selection fragment, each possible target for a given node type, described by the flow graph, shall produce a template edge from the relevant instantiation gap to the instantiation fragment for the given flow graph node and context node.

Any of the above selection fragments can be handled in the presence of predicates: Since the predicates can filter out any nodes, we must insert an edge to an empty template at every instantiation. I.e. we model that any of the nodes can have been filtered out.

According to the mining this takes care of most of the selection expressions. The modular design easily allows for extending the summary graph construction to handling more complicated XPath selections.

Illustrations of some of the selection fragments described above, can be found in Figure 4.17, and Figure 4.18.

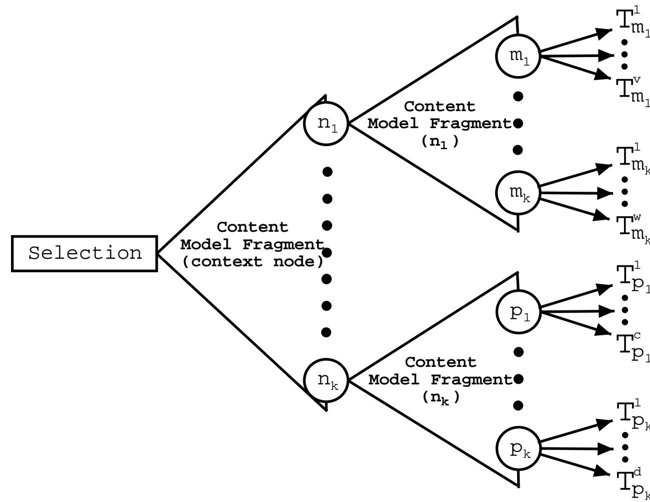


Figure 4.18: Selection fragment for: "`child::`*test*/`child::`*test*".

Regarding unions of location paths, recall that the possible targets for each selection path have been analyzed separately. Since the nodes selected by each path end up in the same node-set, this separate knowledge must now be merged into one. In general, this merging can be hard to do precisely. Consider for example the select expression "`child::n/descendant::n/child::m | child::n/child::m`". Both paths select `m` elements, but different `m` elements. The first path requires loops involving `n` elements, while the second disallows such loops. To model this precisely in the selection fragment, we would need to distinguish `m` elements by whether any loops of `n` elements occurred before reaching `m`.

Fortunately, such complexity in the selection paths are rare, and we shall handle the most frequently used case, while approximating the rest with the worst-case construct. We shall handle a union of single-step paths such as "`child::`$test_1$ | `child::`$test_2$ | `child::`$test_3$", by simply constructing the selection fragment for a single child-step, and then adding instantiation edges to the possible targets for each path. This will be enough in most cases. It certainly handles the general identity selections: "`child::node() | attribute::node()`". Note that the handling of predicates described above applies to these unions as well, though empty instantiation edges only need to be inserted if all the paths able to select the given node type contain predicates.

One final issue that needs to be handled are the `sort` instructions of XSLT. They specify complex orderings, effectively overriding the document order which we have based our selection fragments on. Thus the occurrence

of a `sort` in an `apply-templates` instruction forces us to approximate with the worst-case construct of arbitrary order and cardinality. However, in the special case of only a single node type being selected, the order is insignificant, and we can ignore the presence of `sort` instructions.

An example of an output summary graph, constructed from the news transformation of Figure 2.4, can be found in Appendix A.1.

4.7.2 Summary Graph Inclusion Analysis

With the output summary graph finally constructed, we must test whether all the XML documents it represents are contained in the output DTD. More precisely we want to test whether $\mathcal{L}(SG) \subseteq \mathcal{L}(D_{out})$, where $\mathcal{L}(SG)$ is the set of all unfoldings for SG and $\mathcal{L}(D_{out})$ is the set of all XML documents conforming to D_{out}.

Such an analysis of summary graph validity, with respect to the strictly more expressive schema language DSD2, is examined in the JWIG project [21]. The difference between instance document validity checking and summary graph validity checking of course lies in the fact that we need—at least conceptually—to check all unfoldings of the summary graph.

In our case, checking only for DTD validity, the analysis becomes much more simple. It essentially reduces to the *declaration checking* phase of the DSD2 analysis, where only a single regular expression for legal content must be considered for each element and attribute, and with much simpler constraints on text and attribute values.

The algorithm proceeds as follows: First, all default values are insterted similarly to the *normalization* phase of the DSD analysis. Next, for every element, the summary graph nodes that represent the content of that element are used to construct a *context-free grammar* describing the possible content of that element. This grammar is then approximated by a regular language. Without so-called *top-level loops*[7] in the content-describing summary graph nodes, the grammar actually describes a regular language, and the inclusion test is efficient. Otherwise, some precision is sacrificed in the approximation, in order to allow the more efficient regular language inclusion algorithm to be used in contrast to examining inclusion for the context free language directly. As suggested by the JWIG paper, it is possible to provide much better context-free grammar to regular language approximations. In [59], Mohri and Nederhof examines this topic.

In parrallel to the content examination, the attributes on each element are checked against the attribute list in the DTD. This amounts to checking that (1) each attribute is legal, (2) that required attributes are present, and

[7]Top-level loops are loops in the summary graph, where the gaps in the loop occur only at the top level of the summary graph node templates.

(3) that the attribute values are legal. The latter depends on the string edges in the summary graph. Because of our limited string handling, the string edges are quite easily checked.

The floating attributes in our summary graph model pose the greatest difficulty here. In order to extract the presence of attributes or lack thereof, we shall construct a regular content expression for attributes in much the same way as for the rest of the content. This regular expression now approximates the order and cardinality of the floating attributes. Ignoring the order of the attributes, we shall assign *cardinality labels* to each attribute from the set $\{+, *\}$. We shall do this recursively on the regular expression for the attributes in a bottom-up fashion, thus assigning a map from attribute names to cardinality labels for each sub-expression. The final map for the root expression suitably describes the presence of each attribute. A $+$ means that a floating attribute with the given name is constructed in every unfolding. A $*$ means that an attribute with the given name may or may not constructed.

If an attribute is required on the element being examined, that attribute must have assigned a $+$ label. Otherwise, it may not always be present, and we report an error. Any attribute, which has been assigned a cardinality label, and which is not declared for the element, results in an error report. Finally, each value assigned to some desclared attribute under the element being examined, must be legal according to the DTD.

4.8 Summary

We have now completed the design of our static analysis technique for determining output validity of XSLT 1.0 transformations. The transformations are first reduced to a manageable form called reduced XSLT. The flow analysis then statically tries to determine the flow of template rule instantiations going on when the transformation is executed. Finally, the flow analysis is used to create a summary graph representing all possible output of the transformation. This summary graph is then compared to the given output DTD, in order to determine output validity, or detect possible inconsistencies with such output validity.

However, the question remains whether the analysis technique is useful in practice. We shall examine this question in the following chapter.

5

Experimentation and Interpretation

5.1 Implementation Details

In order to test the ideas presented in Chapter 4, and to prove the practical usefulness of the analysis, we have implemented most of the analysis and run it on a number of examples gathered from various sources, mostly the Web. None of the examples, except our running news example, have been written specifically to test the analysis, so they all represent "practical XSLT" in some form. Details of the analysis which have not been needed in the experimentation have not been implemented. The implementation is not meant as an commercial quality tool.

The implementation is written in Java, which has an extensive and very useful library of algorithms and data structures. More importantly, this choice of language allows us to re-use the summary graph validation algorithm of JWIG in our project, which has graceously been made available by Anders Møller. The main drawback of this is that the JWIG summary graphs are unable to express the floating attributes of our summary graph model. Any floating attributes, which are in the same summary graph template as their parent, can be emulated in JWIG summary graphs by attribute gaps. Those that are at the top level of a template can not be handled unless we start analyzing the summary graph ourselves, i.e. separately from the JWIG package. We have opted instead to analyze this feature by hand in our experimentation. The use of an existing implementation of summary graphs and the associated inclusion analysis saves considerable amounts of time.

An advantage of using the JWIG summary graph inclusion analysis is that it works on DSD 2.0 [60] schemas rather than DTD schemas. DSD 2.0 is strictly more expressive than the DTD language: a DTD can easily be converted to a DSD 2.0 schema, and this is exactly what we do. Provided by Anders Møller is a DTD to DSD schema converter. If a single element is known to be required as document element, this information can be included in the DSD 2.0 schema. DSD 2.0 can not express multiple allowed document elements, but fortunately this is not needed for those of our examples left after sorting out transformations using multiple inputs or extension elements. Note that multiple document elements is relevant for stylesheets operating on for example XSLT documents, or TMML [36] documents. TMML and its associated XSLT transformations was discarded as an example because of multiple input document use.

Another benefit of the DSD summary graph analysis is that it is namespace aware, i.e. we are able to handle and analyze multiple namespaces in the output.

Recognizing the fact that the exactness of the output schema is important for the usefulness of our static output validation guarantees, we shall try to analyze as many of the examples with the more precise DSD schema for XHTML instead of the DTD converted to DSD. This means that our implementation is able to provide higher quality error reports or static guarantees for a number of our examples, but in the cases where spurious errors occurred as a result of the more precise output schema, we have fallen back on the DTD converted to DSD. Such fallbacks typically involves the stringent constraints on URLs in attribute values expressed in the XHTML DSD.

As it turns out, the use of `copy-of` instructions in conjunction with parameters is very seldom used. It is mostly used for copying input data over from alternative input sources, which we do not handle. As such, our parameter analysis has become superfluous. None of our examples make use of it, so it has not been implemented. It shall remain for future work to test the analysis, perhaps in the setting of multiple input sources.

The simplification outlined in Section 4.4 is implemented in a manner so as to minimize the number of passes required. It proceeds bottom up, in order to avoid unnecessary template rule copying due to the template rule splitting. For example, if a template rule contains a `choose` instruction, it makes no difference semantically whether we first split the template rule and then de-nest the `choose` instruction in each of the split rules, or whether we first de-nest and then split. However, the latter approach potentially results in fewer template rules in the simplified document. The bottom up simplification achieves this minimization.

The construction of the JWIG summary graph is quite similar to if we

had our own summary graph implementation. The basic functionality is almost the same. The floating attributes are handled as follows:

- If the parent of the attribute is in the same template as the attribute itself, we replace the floating attribute with an attribute gap with the proper name. An attribute gap is essentially a literal attribute where the attribute value is described by string edges.
- If there is no parent in the same template, we remove the floating attribute and report an error.

Apart from this, the only difference in implementation between the JWIG summary graph model and our own, is that the summary graph templates in JWIG are described with literal content, i.e. with literal elements, text and so on. And as mentioned, the elements and global attributes can belong to differing namespaces.

Optimizations

A number of optimizations have been employed in the flow analysis, some of which have been mentioned in Chapter 4. Most of the time complexity in the flow analysis stems from the large number of possible flow edges examined, and from the constructions of and operations on finite automata. In order to reduce the use of the path automata test, we always run the simulation test first, and discard the edge immediately if the resulting edge flow is empty. Also, we do not bother running the path automata test if the target match expression is a root node selection (i.e. "/"), or a relative path with a single node step. In these two cases, the path desribes nothing but the node type involved, and this is analyzed just as well with the simulation test alone.

In order to bring down the number of prospective edges examined in the first place, we employ a *context node insensitive flow analysis.* This is a reduced form of the flow analysis, where the possible context nodes for instantiation are not propagated along the edges. Instead, each template rule is only analyzed once, where it is assumed that the rule could be instantiated with *any* node type, or at least any node type possible under the match path. Thus, the context insensitive flow propagation tests serve to identify roughly which edges might carry flow, and which can never carry any. The tests themselves remain unchanged in this new setting, but the concatenated selection path is altered so that it does not take the context node into account as follows:

$$CONCAT_{insensitive}(M_s, S) = M_s/S$$

The priority override test is altered similarly by removing the intersection with $\Sigma^* n$. The test then becomes the following:

> *Given edges e_1 and e_2 with the same origin: Let S be the non-downwards-axis stripped concatenation of the match and selection paths of the origin flow graph node. If $priority(e_1) > priority(e_2) \wedge$ e_1 contains no predicates $\wedge$ $FA(match(e_2)) \cap FA(S) \cap FA_{D_{in}} \subseteq$ $FA(match(e_1))$, then edge e_2 is removed.*

The automaton inclusion now determines if *every* path which can occur under D_{in} and which is accepted by the match path of e_2, can also be accepted by the match path of e_1.

The point is now that all the edges which have been discarded in the context insensitive analysis, can be ignored entirely in the context sensitive analysis. This narrows down the number of edges examined considerably. In the most extreme case observed, the number of edge tests was taken from 67,227 to 11,386, which makes it a reduction to a 6th of the original number of edge tests. However, it can go the other way too. In another example, the number of edge tests increased from 1,393 to 1,597, which makes it about 15% more than originally. Nevertheless, the context node insensitive analysis serves to smooth out the possible blow-up from the context sensitivity, and in most examples it improves performance.

Finally, we try to keep down the number of nodes and edges in the output summary graph by skipping superfluous parts of the selection fragments. If only attributes are selected, the modelling of children is skipped, and if only child content is selected, the attributes are skipped. The ubiquitous comments and processing instruction gaps can also be skipped when they are not part of the selection.

Analysis Output

The analysis output centers around the error reports. If no errors are found, the transformation analyzed is guaranteed to be output valid. Otherwise, the error reports describe possible inconsistencies with the output schema. A typical error report can look like this:

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

The error report consists of four items:

- The error message, describing the nature of the inconsistency with the output schema.
- The signature of the involved template rule. This is essentially a printout of the template rule element and its attributes as it appears in

the source document, and it helps the developer to locate the relevant fragment of the transformation.

- The context node is the type of node used for instantiating the template, when the inconsistency occurs.
- The DSD fragment is a piece of the output DSD, describing the attribute name or content model involved. The content model is either shown directly, or a named content declaration from the DSD is referred to. The `x:` namespace prefix shown above is simply the namespace prefix assigned to the given element in the DSD schema. It is in this case bound to the news DTD from our running example.

Not all these pieces of information are relevant to every kind of error reported, so only those relevant will be displayed for the given type of error.
Another example of an output report is:

```
***Validation error: required attribute missing in element 'item'
  Rule:         <xsl:template match="child::*">...</xsl:template>"
  Context node: item
  Element:      <item category="national">...</item>
  DSD:          @date="???"
```

This particular example represents the kind of errors that our incomplete handling of floating attributes can produce. However, it may result from other inconsistencies as well, such as the attribute actually missing. An error often seen in XHTML outputting transformations is missing `alt` attributes on `img` elements. The `alt` attribute is a required, but often forgotten feature of HTML and XHTML.

In order to aid the identification of errors, and to properly test the functionality of our analysis, the implementation outputs intermediate results in the form of two Graphviz [35] *dot* files. One representing the XSLT flow graph generated during analysis, and the other representing the output summary graph. These can be converted to PostScript format and viewed in any PostScript capable viewer. However, the graphs—especially the summary graphs—can become quite large, making them infeasible to examine visually. In such cases, the error reports must suffice. For our purposes of examining the precision of our analysis, later in this chapter (Section 5.3), the smaller examples shall be more than adequate.

When converted to PostScript format, the flow graphs take on the form shown in Figure 4.9. Each node in the illustration is labeled with its priority (made explicit in the simplification phase), match path, and context set. The edges are each annotated with their edge flow.

Ellipses describe flow graph nodes originating from user template rules. Boxes (not seen in the news flow graph) are rules generated in the simplification phase, such as from de-nesting templates for `choose`, `if`, `for-each` and similar. The diamonds represent the built-in template rules.

The summary graph *dot* file can be output in two different forms. One, with the same labels as on the flow graph nodes, except that only a single context node type is given. The other form shows the exact XML template of each summary graph node. These summary graph files are—at least on the smaller examples—well suited for inspecting the result of the summary graph construction.

The implementation can be downloaded from the thesis Web site, at:

`http://www.daimi.au.dk/~madman/xsltvalidation/`

Also on the site is instructions for running the implementation, and all our example transformations and their relevant schemas.

5.2 The Examples

The trouble with gathering examples for experimentation have mainly been that not only the transformations themselves, but also schemas for the input and output had to be found. These are not so often available as one might think. Constructing the schemas ourselves would be tedious work, and would not yield quite the same realism. In order to get a decent amount of example transformations with accompanying schemas, we chose to gathering also transformations with only example XML documents available. A example document can be used to automatically produce a DTD through the SAXON DTDGenerator [49]. The resulting DTDs will be very *tight* around the example document, i.e. it will accept the XML document itself, and other documents very close to it in structure. This should, at least for input schemas, be a safe approximation of the original idea of the authors. Going through this tool also makes the schemas independent of our hands, so that the examples can not be formed to our needs.

Not surprisingly, most of the examples found produce HTML or XHTML output. The problem with this is that XHTML is quite loose in its requirements. Most elements are allowed as children of each other in arbitrary number in one large jumble. It is, however, a dominant use for XSLT, so it only makes sense to examine how the analysis performs on such transformations. Still, we shall examine transformations of non-XHTML output with particular interest when it comes to testing the precision of the analysis.

After gathering a number of examples from various sources independent of this project, those examples that included features not supported by our analysis, such as multiple input or extension elements, were sorted out. Features like multiple namespaces in the input have, instead of being sorted out, been reduced to a single namespace, in order to have more examples to run

Example	Input DTD	Output Schema
poem.xsl (35 lines)	poem.dtd (8 lines)	xhtml.dsd (2,278 lines)
AffordableSupplies.xsl (42 lines)	Catalog.dtd (31 lines)	xhtml.dtd (1198 lines)
agenda.xsl (43 lines)	agenda.dtd (19 lines)	xhtml.dsd (2,278 lines)
news.xsl (54 lines)	news.dtd (12 lines)	xhtml.dsd (2,278 lines)
CreateInvoice.xsl (74 lines)	PurchaseOrder.dtd (37 lines)	dtdgen.dtd (32 lines)
adressebog.xsl (76 lines)	dtdgen.dtd (22 lines)	xhtml.dsd (2,278 lines)
order.xsl (112 lines)	order.dtd (31 lines)	fo.dtd (1,480 lines)
slideshow.xsl (118 lines)	slides.dtd (26 lines)	xhtml.dtd (1,198 lines)
psicode-links.xsl (145 lines)	links.dtd (15 lines)	xhtml.dtd (1,198 lines)
ontopia2xtm.xsl (188 lines)	tmstrict.dtd (113 lines)	xtm.dtd (202 lines)
proc-def.xsl (247 lines)	proc.dtd (69 lines)	xhtml.dtd (1,198 lines)
email_list.xsl (257 lines)	dtdgen.dtd (41 lines)	xhtml.dtd (1,198 lines)
tip.xsl (262 lines)	dtdgen.dtd (56 lines)	xhtml.dsd (2,278 lines)
window.xsl (701 lines)	dtdgen.dtd (84 lines)	xhtml.dtd (1,198 lines)
dsd2-html.xsl (1,353 lines)	dsd2.dtd (104 lines)	xhtml.dsd (2,278 lines)
xhtml2fo.xsl (1,697 lines)	xhtml.dtd (1,198 lines)	fo.dtd (1,480 lines)
xmlspec.xsl (2,528 lines)	xmlspec.dtd (2,561 lines)	xhtml.dtd (1,198 lines)
identity.xsl (9 lines)	-	-

Table 5.1: The examples we use in our experiments and their input and output schemas.

the analysis on.

Also a certain amount of pre-processing was done on the examples. Conversion from XHTML to HTML output was performed with an XSLT transformation, and unnecessary use of `disable-output-escaping` was converted to less ugly constructs. And various other disagreements with the XSLT specification—which is unimportant for our experimentation—were corrected as well.

For those examples that produce HTML output (i.e. which use the HTML output method), we ought to have produced a DTD describing the structure of HTML documents, since it is slightly different from XHTML. This detail has been ignored however, as it is not of importance with respect to testing our analysis.

An overview of the examples we shall examine can be found in Table 5.1, sorted by the size of the transformations in lines of code. Note that the sizes of both transformations and schemas are before entity expansion. The `xhtml.dtd`, and `xhtml.dsd` are the XHTML 1.0 DTD and DSD schemas respectively. `fo.dtd` is a DTD for the XSL Formatting Objects (XSL-FO), which are a part of the Extensible Stylesheet Language (XSL) specification (recall Section 2.3). The rest of the DTDs are less widely known. They can all be found in versions modified for this project, as described above, together with the transformations and output schemas at the thesis Web site (`http://www.daimi.au.dk/~madman/xsltvalidation/`).

The examples perform various tasks, but mostly they produce an HTML/ XHTML presentation of XML data, or convert between two XML classes.

The CreateInvoice.xsl example has extraordinarily an automatically generated output DTD. The tightness of this DTD is directly responsible for several of the errors found in the transformation: the `currency` attributes in the input schema are declared as `PCDATA`, while the DTD Generator decided it should be of type `NMTOKEN`. When the values are copied over from the input, it inevitably results in errors. This is to be expected, and while it may be unfair to the authors of the transformation, the issue is not of importance to our testing of the analysis.

5.3 Precision

Let us start by examining our running news example from Figure 2.4. The flow graph constructed in the analysis can be found in Figure 4.9. The analysis produces no errors on the news example, so let us inspect the analysis run.

Firstly, we can note that the built-in templates have been overridden in all but one case: for the root element. Indeed, an explicit rule for matching the root node is missing in the transformation. We can also see that the `match="p[1]"` template rule has not overridden the `match="p"` rule, even though it has higher priority. This is a good example of why we must test for the existence of predicates in the priority override filter. The `match="p[1]"` rule only matches *first* `p` children of `text` nodes. The rest is handled by the `match="p"` rule. Thus, both rules recieve some of the `p` nodes, and both must be present in the flow graph. We can conclude that the priority override filter seems to work well. Apart from this, every `apply-templates` selection produces some flow into other template rules. The `value-of` instructions do not result in recursion, so they have no part in the flow graph. We shall claim that the flow analysis has completely exposed the flow of node types for template instantiation in the news transformation.

An illustration of the output summary graph generated for the news example can be found in Appendix A.1. The automatically output summary graph can not be fit into a page, but the illustration in the appendix has been manually constructed from the automatically generated output. The major difference is that the selection fragments have been made much more compact. Also, the illustration has been made so that it resembles the flow graph illustration closely. This should make the transition from flow graph to summary graph clear.

To test that the generated output summary graph really represents the output of the transformations, we can try infusing errors in the transformation. Inserting an `<li/>` element[1] anywhere in the transformation, produces a "sub-element 'li' (http://www.w3.org/1999/xhtml) of element 'body'

[1] `li` elements describe list items in XHTML. The element is allowed as child only of `ul`, `ol`, `menu`, and `dir` elements.

(http://www.w3.org/1999/xhtml) not declared" error. Thus, all the template rules must be represented in the summary graph. Also, removing the required element `title` under the `head` element gives us the following error: "contents of element 'head' (http://www.w3.org/1999/xhtml) does not match declaration". A number of other errors such as invalid elements or attributes can be infused as well, producing suitable error messages, and a visual inspection of the summary graph does not reveal any problems. We conclude that the summary graph must be describing all possible output of the news transformation, and apparently precisely enough to statically guarantee output validity of the transformation.

Recall that one of our goals was to be able to analyze the general identity transformation. This has been achieved to the extent possible under our limited floating attribute handling in the implementation. What this means is that running our analysis on the identity transformation inevitably fails on required attributes in the output schema. Basically, every attribute in the input DTD is constructed with an `attribute` instruction (representing the simplified form of the `copy` instruction). Each of these attribute instructions are contained in a separate template rule, and do therefore not have a parent in the same template in the output summary graph. The result is that none of the attributes are represented in the output summary graph, and every attribute required to be present without having a default value, is therefore responsible for a spurious error report. However, this is solely an effect of our inadequate implementation. The analysis design itself would be able to recognize the attributes.

The flow graph for the identity run on the news DTD can be found in Appendix A.2. Recall that `copy` instructions are replaced by `apply-templates` instructions in separate modes in the simplification phase. In this case, the mode is `auto_0_copy`. The `auto_0_copy` templates take care of copying the given node, as well as contain the contents of the original `copy` instruction, which in this case means a simple recursion on all children and attributes.

An interesting example of how our analysis exposes the structure of the output, on a fragment of the `ontopia2xtm.xsl` transformation, can be found in Appendix A.3.

Practical examples

Let us now examine on a more general level, how the analysis performs on the various examples gathered from independent sources. Each example has been run, and the error reports closely examined and categorized, except for the two biggest examples: `xhtml2fo.xsl`, and `xmlspec.xsl`. These two produced too many errors to go through in detail. We have also left out the error reports resulting from the inadequate floating attribute handling of our implementation, though only two such errors occurred in the independent

Example	Correct Errors	Spurious 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	1
psicode-links.xsl	20	0
ontopia2xtm.xsl	0	6
proc-def.xsl	6	1
email_list.xsl	3	0
tip.xsl	1	1
window.xsl	0	22
dsd2-html.xsl	0	0
identity.xsl, news.dtd	0	0
identity.xsl, fo.dtd	0	0

Table 5.2: The number of errors generated in the analysis of each example. Errors resulting from the lacking analysis of floating attributes have been ignored.

examples. Table 5.2 shows the number of correct and spurious errors on each of the example transformations. The correct errors are those that were determined to describe actual inconsistencies with output validity, while the spurious errors are those that were determined not to be a problem in actual executions of the transformation.

In the particular case of the `dsd2-html.xsl` example, a lot of the input data to the transformation is defined in additional namespaces not described in the DTD schema for DSD 2.0. Thus, the result of the analysis is an output validity guarantee, given that the input to the transformation conforms to the official DSD 2.0 specification.

Correct Error Reports

To our satisfaction—as the table in Table 5.2 shows—a considerable amount of errors were found in the example transformations analyzed. The errors found range over a number of different kinds of errors:

- Misplaced elements, such as `link` elements occurring outside the XHTML header, where they are required to be.
- Undefined elements, attributes, and attribute values.
- Missing elements or attributes. A typical missing attribute already mentioned is the `alt` attribute of `img` elements in XHTML, and also the `title` element in the XHTML header, as well as the header itself, has been seen missing.

- Empty content where it is not allowed. Not unexpectedly, a typical example is empty lists in XHTML. The list elements of XHTML, such as `ul` and `ol`, require at least one list item element `li`, but it is often the case that such lists can in fact become empty. This error was also observed and worked around in the JWIG project. They simply remove empty lists from the output, which is convenient when using the language. XSLT does no such thing though, so the empty lists are output validity errors.

- Wrong namespaces. The ability to catch these errors arise from our usage of the DSD 2.0 analysis package, and surprisingly one of the example transformations turned out to contain a number of these errors: `psicode-links.xsl`. They all come from copying elements directly over from the input, without realizing that—in the given case—the namespace has to be changed as well.

- ...

Most of the errors found are easily identified and corrected. However, some of them are more tricky, such as the "sub-element 'li' of element 'body' not declared" error in the `proc-def.xsl` transformation. The construction of the `body` element is quite complex, and the `li` element in question occurs at a recursion depth of four template instantiations inside each other. It turns out that the template rule generating the `li` element appears twice in the transformation document, with the exact same signature: `<xsl:template match="execlist/proc">`. This clearly indicates that something is amiss, since two equally suitable matches on a selected node at runtime is an error, according to the XSLT specification. Such errors are not the focus of this project though.

Spurious Error Reports

The spurious errors describe invalid output that can in fact never occur at runtime. The frequency of these errors can be said to measure the precision of our analysis. At first glance, there seems to be an awful lot of spurious errors in the examples analyzed. The main focus of our analysis have been on exposing the structure of documents, and as we shall see, the spurious errors coming from inadequate structural analysis are sparse. The rest of the errors are a testament to the fact that analysis of structure alone is not quite enough. It seems that strings must be analyzed more carefully than we have done.
The spurious errors found in the examples can be categorized as follows:

Spurious Error Category	**Number of Errors**
Inadequate Structural Analysis	3
Inadequate String Analysis	30

As can be seen, there are three spurious error reports with a structural nature. The first of them is from the `tip.xsl` example. The error report occurs because of inadequate summary graph construction for rhe selection path `"../analysis"`. In our analysis, this selection results in the worst-case selection fragment expressing "arbitrary order and cardinality", but inspecting the input DTD reveals that this selection will in fact always yield exactly one element. Unfortunately, this knowledge is required in order to satisfy the output DTD requirements of at least one child element in the given situation.

The issue in this particular case could easily be handled by "walking the selection path" on the input DTD and observing the cardinality in each step. However, precisely analyzing upwards paths in general is not quite that easy, as several different node types can be selected in each step. But again: Our analysis could easily be extended to handle the simpler upwards selections such as the one causing a spurious error report in the `tip.xsl` example.

Regarding the last two spurious errors of structural category, they are both located in the `CreateInvoice.xsl` example transformation, and they both stem from selections of the type `"//a"`. Before considering how to handle such selections, we note that they are in fact entirely unnecessary in this particular example transformation. All the `"//a"` selections in `CreateInvoice.xsl` could be replaced with simple child selections like `"a"`, without loss. Nevertheless, such selections do occur regularly, and could end up resulting in spurious error reports less easily avoided.

Selections of the form `"//a"` represent simply any `a` elements in the document, and they can be modeled by inserting a summary graph fragment representing the entire structure of the DTD, much like the individual content models are handled in our analysis. However, such fragments can become very large, and some minimization will be essential so as to not blow up the size of the summary graph considerably.

As shown in the error categorization table above, the rest of the spurious errors are a consequence of inadequate analysis of string-values in the transformations. Some of these errors come from copying the value of an input attribute over to an output attribute. If these both are of type `NMTOKEN` for example, then only name-characters can occur, and the value of the output naturally must always be valid. Our analysis is unaware of the difference in character sets, and models the output attribute value as any string. But this includes all charaters, and is not always an `NMTOKEN`. Also, an attribute constructed as `id={generate-id(...)}` will always be of type `ID`, but again the analysis will model it as any string even though the output may allow only `ID` values. Handling such simple cases as described here would avoid many of the spurious errors found in our example set, and would thus greatly benefit our output validation analysis.

There are however some string related errors which would be much harder

to handle. In particular, one of the spurious errors in `ontopia2xtm.xsl` is part of a template calling mechanism where a tokenized attribute value (`IDREFS` to be specific) is split up with the XPath expression `"substring-after($n_topicRefs,' ')"`, and a template instantiation called for each token. Constructs like this, will be inherently difficult to analyze. But we can note that the analysis could in fact easily be made to validate the transformation by removing a superfluous `if` instruction, so in case the author of the transformation wished to achieve static output validity guarantees, he could still easily get them here by performing small modifications in the transformation.

Conclusions

Summing up, our analysis seems to do well with exposing the structure of output, but insufficient string handling result in some amount of spurious error reports. This is an area that can be improved on, and it ought to be possible to avoid most of the spurious errors found here with little effort. Overall, the analysis seems to do well with exposing errors in practical XSLT stylesheets.

5.4 Performance

An important aspect of practical usefulness of our analysis is that it performs well enough to be used in day-to-day development of XSLT. In order to test the performance of our analysis, we have measured the execution time for the analysis on each of our examples from Table 5.1. The results can be found in Table 5.3, together with the number of nodes and edges (in that order) in the constructed flow and summary graphs. The tests were performed on a 3 GHz Pentium 4, with 1 GB RAM, running Linux. Each of the main phases, i.e. the flow analysis, summary graph construction, and summary graph inclusion analysis, were timed individually. The total time measures all aspects of the analysis, including loading the files from disk, simplifying the transformation and so on.

As the table shows, most of the smaller examples are executed within a few seconds, which can only be said to be satisfactory. However, the performance does seem to worsen quite a bit with the larger transformations. But—as the identity transformation runs illustrate well—the size of the transformation is not the only important factor on the performance. Note that the identity runs are sorted by the size of the DTDs in number of lines, but apparently the number of lines far from reflect on the execution times. The picture becomes much clearer if we examine the number of elements and attributes in each DTD as depicted in Table 5.4. Especially the number of attributes in each DTD differ quite a bit, and the number

Example	XFG Size	SG Size	Flow Analysis	SG Construction	SG Analysis	Total
`poem.xsl`	10/16	37/58	0.221 sec	0.072 sec	0.046 sec	0.927 sec
`AffordableSupplies.xsl`	2/2	5/17	0.049 sec	0.045 sec	0.281 sec	1.066 sec
`agenda.xsl`	5/5	14/24	0.081 sec	0.059 sec	0.076 sec	0.825 sec
`news.xsl`	10/11	35/46	0.184 sec	0.075 sec	0.065 sec	0.918 sec
`CreateInvoice.xsl`	12/13	39/61	0.252 sec	0.112 sec	0.859 sec	1.773 sec
`adressebog.xsl`	10/23	180/232	0.187 sec	0.203 sec	0.316 sec	1.319 sec
`order.xsl`	10/21	73/100	0.257 sec	0.113 sec	0.158 sec	1.172 sec
`slideshow.xsl`	18/33	99/155	0.360 sec	0.138 sec	0.817 sec	2.108 sec
`psicode-links.xsl`	20/50	117/187	0.424 sec	0.153 sec	0.188 sec	1.454 sec
`ontopia2xtm.xsl`	37/45	140/178	0.336 sec	0.196 sec	0.826 sec	2.082 sec
`proc-def.xsl`	21/22	155/189	0.372 sec	0.186 sec	0.801 sec	2.102 sec
`email_list.xsl`	23/38	111/180	0.391 sec	0.176 sec	0.345 sec	1.693 sec
`tip.xsl`	42/71	209/283	0.691 sec	0.240 sec	0.282 sec	1.916 sec
`window.xsl`	45/55	182/333	0.407 sec	1.467 sec	3.017 sec	5.831 sec
`dsd2-html.xsl`	167/245	26,515/46,184	6.946 sec	15.224 sec	56.168 sec	79.553 sec
`xhtml2fo.xsl`	217/1,275	10,421/17,655	258.838 sec	6.428 sec	12.771 sec	279.774 sec
`xmlspec.xsl`	318/2,101	12,182/24,267	150.057 sec	6.959 sec	89.314 sec	247.691 sec
`identity.xsl, news.dtd`	9/10	67/90	0.155 sec	0.155 sec	0.118 sec	0.819 sec
`identity.xsl, dsd2.dtd`	9/10	1,200/2,653	0.439 sec	1.120 sec	1.408 sec	3.515 sec
`identity.xsl, xhtml.dtd`	9/10	10,144/15,966	14.455 sec	6.365 sec	3.398 sec	25.035 sec
`identity.xsl, fo.dtd`	9/10	44,802/45,742	581.491 sec	25.563 sec	7.318 sec	615.315 sec
`identity.xsl, xmlspec.dtd`	9/10	4,822/7,459	4.427 sec	3.511 sec	2.598 sec	11.351 sec

Table 5.3: Lists for each example: The sizes of the flow and summary graphs, the execution times of the individual phases (flow analysis, summary graph construction, and summary graph inclusion analysis), and the total execution time.

DTD	Lines	Elements	Attributes	Unique Attribute Names
`news.dtd`	12	5	3	3
`dsd2.dtd`	104	40	33	15
`xhtml.dtd`	1,198	89	1,609	119
`fo.dtd`	1,480	56	10,972	334
`xmlspec.dtd`	2,561	162	563	56

Table 5.4: The number of elements and attributes in a selection of the DTDs.

of attributes appear to be quite independent of the number of lines in the DTD document. This is a direct consequence of heavy use of entities in the DTDs. When many elements share the same attribute definitions, they will typically be specified in entities, and this can apparently blow up the number of attributes greatly. This is taken to extreme hights in the `fo.dtd` schema. The last column in the table shows the number of unique attribute names in each DTD, and these numbers seem more in proportion with the size of the schema.

The conclusion we draw from this discussion must be that it is impractical to handle attributes on each element separately, as we have done in our analysis (recall the $\mathcal{A} \times \mathcal{E}$ attribute set). One should perhaps focus more on the number of distinct attribute definitions in the DTD. This would lie much closer to the number of unique attribute names shown in Table 5.4, and would avoid this blow up in numbers from entity use.

Examining the execution times in Table 5.3 in light of this realization, offers a plausible explanation of why the flow analysis of `xmlspec.xsl` is performed in almost half the time of the flow analysis for the `xhtml2fo.xsl` example. While both the transformation and the DTD in the former example is notably larger than the latter, the number of attributes in the `xmlspec.dtd` DTD is about a third of the number of attributes in `xhtml.dtd`.

Where the input DTDs have great influence on the analysis execution, the output schema size seems to be of little importance. Most of the examples—small and large—are run with large output schemas without the execution time being notably affected.

While testing our analysis, a potential problem in the summary graph inclusion test was uncovered. When the content expressions for some element, constructed in the summary graph inclusion analysis, becomes too large, the process of making the respective finite automaton deterministic can take exponential time: a state is potentially produced for each possible subset of the set of states in the non-deterministic automaton. It turns out that the content expressions generated for output summary graphs for XSLT transformations can become quite large, potentially resulting in a great slow down of the summary graph inclusion analysis. However, in many cases, these complex content expressions arise where the allowed children may oc-

cur in any order and cardinality (i.e. for output content models allowing $(n_1 \mid n_2 \mid \cdots \mid n_m)*$). In such cases, an exact content expression describing order and cardinality of elements is completely unnecessary. All that needs to be checked is, that only allowed node types occur. But, in the case of a more complex output content model, we might have to try and recognize the situation and approximate the content expression, in order to avoid the heavy automaton determinization process. Fortunately, no such case occurred in any of the examples we tested on.

One final consideration is the memory consumption of our analysis. The consumption has been tested with the java profiling tool: JProfiler [34]. Among the three biggest of our example transformations, `dsd2html` consumed the most memory. For this transformation, the maximum heap size allocated by the Java virtual machine is 136 MBs. This is not a precise measure of the memory consumption of our analysis, due to the extra memory required for the garbage collection to function. But it is a safe upper bound. About a 13% of the memory is allocated in the initial loading phases and during the flow analysis. The rest is spent in the summary graph construction. Contemporary desktop machines typically have considerably more memory than 136 MBs, so this seems acceptable. Note also that the `xhtml2fo` and `xmlspec` examples consumed only 80 MBs and 64 MBs respectively.

5.5 Summary

We have implemented our summary graph technique and performed a number of experiments on XSLT transformations, most of which were written independently of this project. The results indicate inadequate handling of string values, but overall the analysis ought to be useful to XSLT developers. Also the execution time and memory consumption of the analysis seems to be reasonable.

6 Conclusions

At the end of our path, the question is: did we reach our goals? Did we solve the problem we set out to solve? In this section, we shall evaluate our efforts and results, and we shall try to establish how our results could be further improved and built upon.

6.1 Evaluation

In recapitulation, we have designed a static output validation technique for statically determining whether an XSLT 1.0 transformation always produces valid output under a given input and output DTD. The analysis builds upon an analysis of the flow of context nodes for template instantiations. This flow analysis manifests itself as a flow graph, which is then used to construct a summary graph, representing a conservative approximation of all possible output of the transformation. Our summary graph model is a slight variation over the summary graph abstraction of the JWIG and Xact projects. The transformation can now be guaranteed to be output valid, if every unfolding of the summary graph is valid under the given output DTD. This can be determined by examining whether the contents of each element in the summary graph will always fulfill the given content model and attribute list of that element. If the transformation can not be determined to be output valid we produce error reports, which should help the developer in locating errors in the transformation.

Our intention with this project was to design and implement a static analysis technique, which is able to guarantee output validity of XSLT 1.0

transformations. Recalling Section 3.4, our goal was for the analysis to be practically useful by being: (1) precise, (2) fast, and (3) by being able to handle most XSLT 1.0 transformations. The last requirement is fickle in the sense that determining what "most XSLT 1.0 transformations" is, or similarly what "practical XSLT" is, is not an easy task. However, we shall claim that our analysis is able to handle at least a large part of practical XSLT. The major obstacle in this matter is the use of multiple input documents. It shall be no secret that a not insignificant part of the example transformations gathered for our experimentation, were sorted out precisely because they used multiple input documents. There is room for improvement on this issue, but still, most examples remained and were part of our experimentation described in the previous chapter.

Regarding the precision of our analysis, we determined on the one hand that the flow analysis was able to expose the structure of template instantiations in transformations quite precisely. On the other hand, we also found out that our handling of strings, particularly in attribute values, were less successful. But overall the analysis was able to detect a considerable amount of errors in the example transformations, and it should be helpful in both debugging transformations and in establishing output validity guarantees.

Finally, the performance results were quite good for most of the examples. However, a couple of situations were exposed, where the analysis can suffer a notable performance hit. When entities are used in describing attributes shared between many elements, the number of attributes we have to handle in our analysis can become excessively large. This is an effect of the way we represent attributes for each element separately. It should be possible to eliminate this particular problem by treating attributes, which are defined identically, as one. The other performance issue manifests itself through unstable execution time of the summary graph inclusion analysis. It is a direct consequence of the fact that determinizing the finite automata, representing the content of some element in the output summary graph, takes exponential time in the worst case. This is not easily handled, but can in many cases be worked around by recognizing unnecessary determinization operations. Perhaps other heuristics can be applied to further bring down these performance fluctuations. Otherwise, we might have to approximate away the troublesome content expressions.

Some additional conclusions can be drawn from our work, which might be of interest. Firstly, the predicates in selections and match patterns apparently have little influence on output validity under DTDs. No predicates were involved in the error reports, spurious or correct, found during our experimentation. Almost the same can be said about the branch tests in `if` and `choose` instructions, which we ignore entirely as well. They were only observed being part of one of the spurious errors, and that was the most complex of the string natured errors where a tokenized string is split up and

recursed on. Thus, this area of our analysis does not seem to be needing much improvement.

6.2 Contributions

The following contributions are made to the field of XML related research:

- A static analysis technique for determining output validity for XSLT transformations, enabling debugging of XSLT transformations and, in many cases, static guarantees of output validity under the given output DTD.
- A flow analysis technique, which exposes the flow of context nodes for template rule instantiation in XSLT transformations. This is used to construct a summary graph describing all possible output of a transformation.
- A variation over the summary graph model, and companion inclusion analysis, with attributes as first class members of XML templates. This is needed in order to model the complex attribute constructions of XSLT.
- An implementation of the output validation analysis, which handles most features of the analysis and successfully analyzes and finds errors on XSLT transformations written independently of this project. The implementation allows visual inspection of the flow and summary graphs produced during the analysis.
- Experimental evidence, which indicates the practical usefulness of our analysis technique in both locating errors in transformations, and in establishing static guarantees of output validity.
- The outline of a simplification technique for reducing XSLT transformations to a more manageable form. The reduced form is described by a short grammar, and it is this reduced form that our analysis operates on.
- An extension of the DTD content models, which simplifies information extraction from a DTD, when underlying the XPath data model.
- Two location path compatibility tests, which conservatively approximates whether a pair of location paths can select or accept the same types of nodes. One test handles all the XPath axes to some degree. The other test is limited to non-upwards axes, but it is more precise, since it examines not only the node types, but also the paths used to reach the nodes. These tests form the core of our flow analysis.

- A priority override test, which conservatively approximates whether one template rule always overrides another for some selectable node type. This test is also vital for the precision of our flow analysis.
- An algorithm for constructing finite automata, which describes all possible top-down paths acceptable by some location path. A companion algorithm constructs an automaton describing all possible top-down paths in a given DTD.
- Some data mining, which suggests how the XSLT language is used in practise.

6.3 Future Work

Firstly, some direct improvements can be made on the analysis. In particular, a more precise string analysis will greatly benefit the analysis. Also, multiple namespaces in both input and output turned out to be often used. Coupling input and output elements with namespaces should be enough to handle this, and it should be incorporated easily in the analysis. However, such a change requires namespace aware input and output schemas, ruling out the DTDs. Lastly, the use of multiple input documents in a transformation is by no means a rare occurrence. Handling multiple inputs in themselves is not a big problem. It simply entails loading a schema for each input, and keeping track of which data comes from which source. However, these extra inputs, and fragments of them, are often handled through variables and parameters. Thus it is likely that some extra attention in this area will be needed as well.

Recalling the discussion in Section 3.2, we note that multiple schemas for a single data source, as explained, is easily handled by a separate run of the analysis with each schema. We can now extend this to multiple schemas for the output as well, simply by running the summary graph inclusion analysis once for each output schema. If just one of the schemas validate the output, then the transformation is output valid for the union of the output schemas. However, the analysis time complexity will be blown up seriously, the more schemas that have to be tested for both input and output. Especially so in the presence of multiple document sources, as all combinations of input schemas have to be considered. In such a scenario, it will perhaps be more feasible to design a generalized schema model, which is closed under union, and then analyze once with the unioned schemas instead.

One of the troubles working with XML and the Web is that all the tools, languages, and standards are in constant evolution. Work on a successor to XSLT 1.0 has been underway for some time. First as XSLT 1.1, and later moving to XSLT 2.0. But also XML, XML Namespaces, and XPath, all of which lay the foundation for XSLT, are moving on. This makes our results

age rather quickly, but hopefully, most of our results will carry over, or at least be of use to the next generation of standards.

One feature of XSLT 1.1/XSLT 2.0 that has already been adopted by XSLT processors is the `document` construct, which allows multiple output documents to be constructed. Thus, this feature already seems to be in use. This feature should be easily incorporated into our analysis, by generating an output summary graph for each output document. Other features of XSLT 2.0, such as being able to further manipulate the results of template instantiations, might not be as easily handled. XSLT 2.0 is turning into a very different language.

While our analysis was developed specifically with static output validation of XSLT in mind, our results could perhaps be of use in other contexts. The issue of optimization comes to mind. Inspecting the flow graphs generated by our analysis, there is often only a single possible target template rule for each `apply-templates` selection, or at least a single target for each given node type. This knowledge will enable an XSLT processor to completely skip finding the target for such a selection node, if it is given that the input to the transformation conforms to the input DTD. The only possible target template is already known, and if the match expression is sufficiently simple, it need not even be inspected. Such optimizations ought to improve runtime performance considerably for XSLT processors. This is especially relevant for transformations which are executed often, such as schemas presenting XML data in XHTML format on some Web site. The analysis time has to be worth the effort.

Our techniques and ideas might also be useful in analyzing other languages, such as XML-QL [31] and XQuery [9]. In particular, the summary graph abstraction has now proven useful outside its original domain, so it seems natural to try and apply it in yet other settings of XML manipulation.

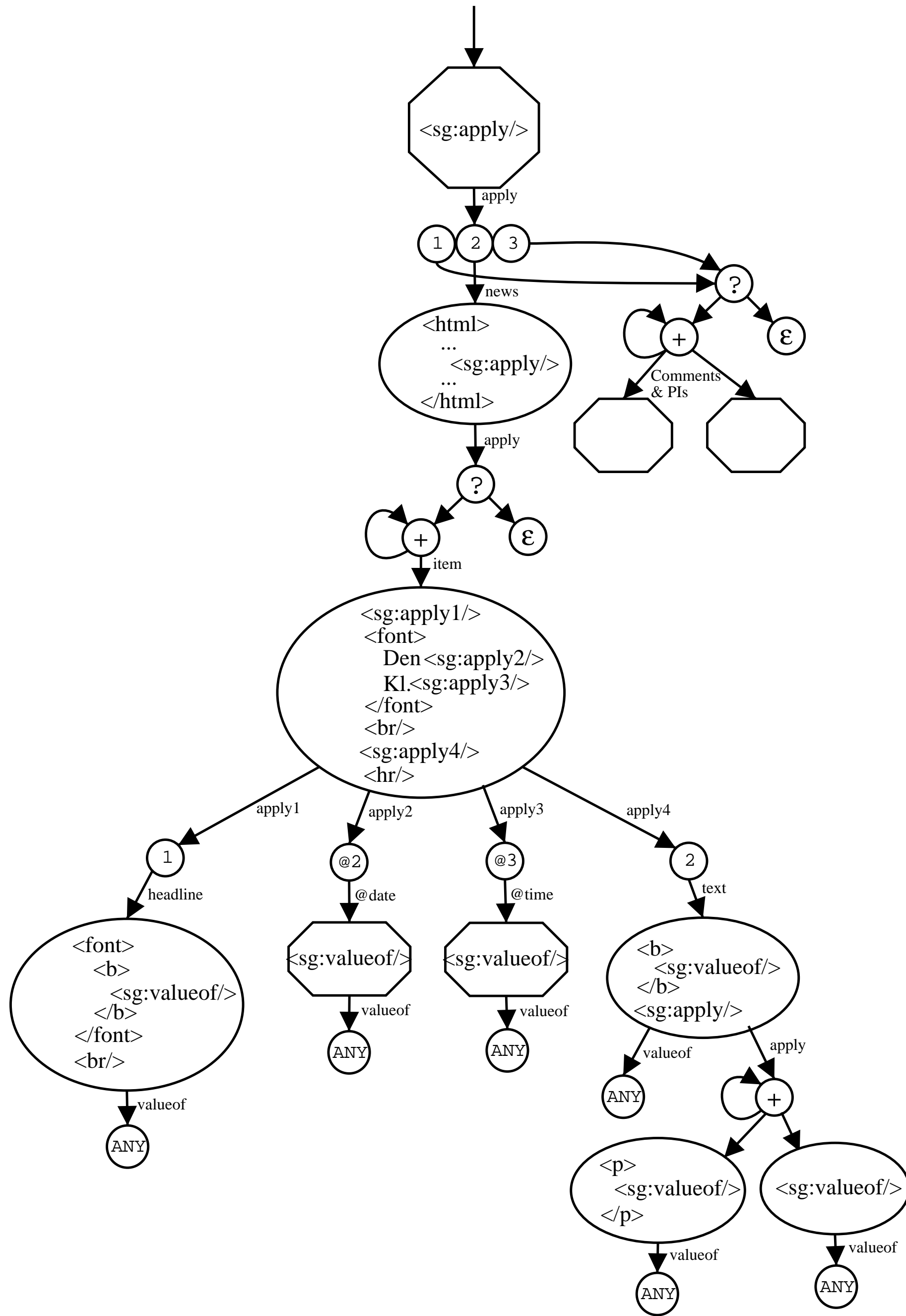
Finally, it would be interesting to try and detect some of the flow errors examined by Dong and Bailey [32]. With our more detailed flow analysis, we should be able to do better.

A

Appendix

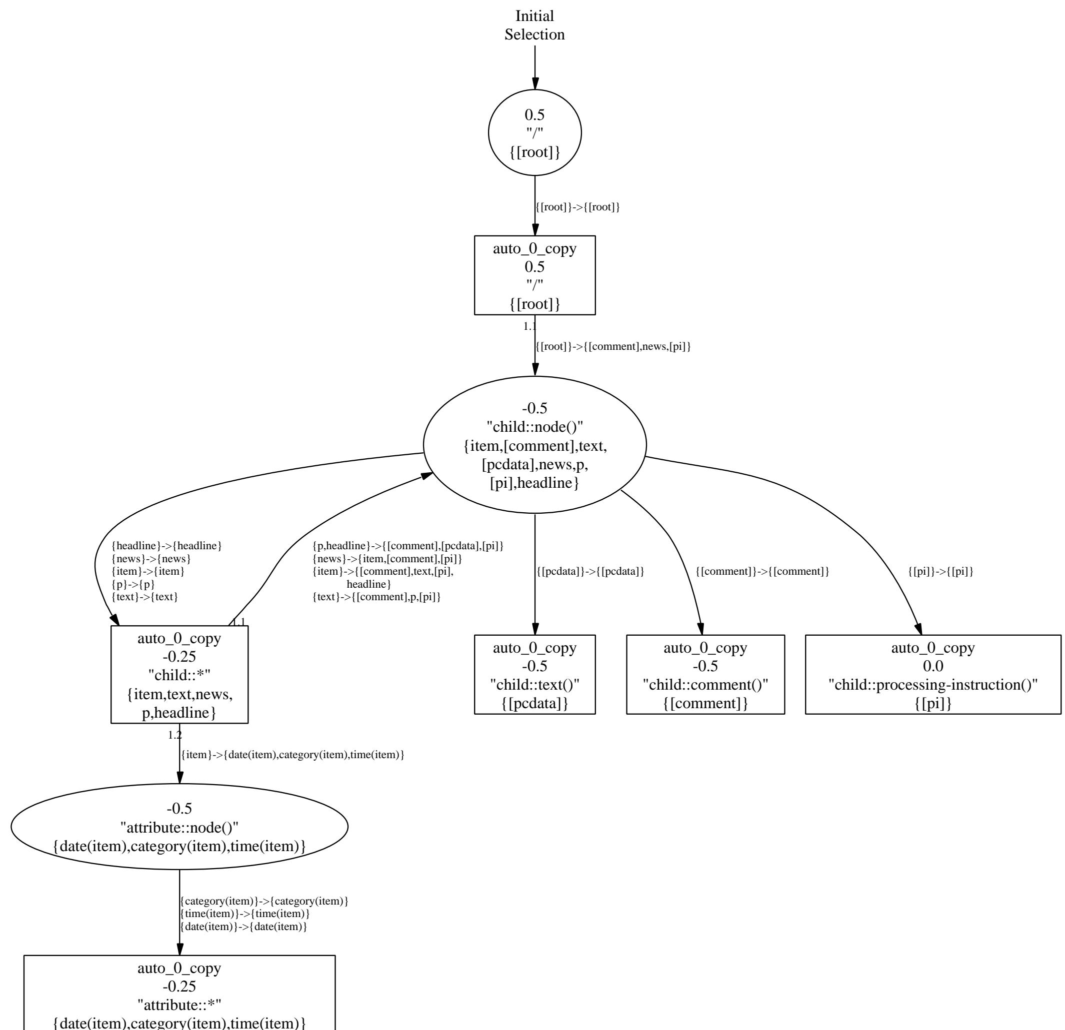
A.1 Output Summary Graph for the News Transformation

Below is a compact form of the summary graph constructed by our analysis, for the news example of Figure 2.4.



A.2 Flow Graph for General Identity on the News DTD

The flow graph constructed by our analysis on the general identity transformation of Figure 3.1 with input conforming to the news DTD of Figure 2.2 can be illustrated as follows:



A.3 A Fragment of the ontopia2xtm.xsl Example

Here we present a good example of how our static output validation exposes the structure of the output of a transformation. The fragments are taken from the `ontopia2xtm.xsl` example, and the transformation code is annotated with small type expressions, representing the content expression which the summary graph inclusion test will extract for the `association` element. A content expression is determined for each XSLT instruction under the `association` element, as well as for the two template rules which take part in the description of the content. The final content expression becomes $((instanceOf \mid instanceOf), scope?, member+)$, which it should be clear is included in the content model for the `association` element: $(instanceOf?, scope?, member+)$. Thus, our analysis correctly produces no content errors on this element.

Note that the floating attributes give rise to another content expression, which we have left out for simplicity. It is not hard to see that the two `attribute` instructions in the fragment below will result in the expression $(@id \mid @id)$, which is legal under the output DTD. Although our implementation is unable to recognize this, it will not produce a spurious error, since the `id` attribute is only optional.

From the input DTD:

```
<!ELEMENT assoc (assocrl+)
<!ATTLIST assoc
          linktype CDATA #IMPLIED
          scope IDREFS #IMPLIED
          type IDREF #REQUIRED>
```

From the output DTD:

```
<!ELEMENT association (instanceOf?,scope?,member+)
<!ATTLIST association id ID #IMPLIED>
```

From the transformation itself:

```
<xsl:template match="assoc">
  <association>
    <xsl:choose>                                    ->()
      <xsl:when test="@id">
        <xsl:attribute name="id">
          <xsl:value-of select="@id"/>
        </xsl:attribute>
      </xsl:when>
      <xsl:otherwise>
        <xsl:attribute name="id">
          <xsl:value-of select="generate-id(.)"/>
        </xsl:attribute>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:apply-templates select="@type"/>           ->(instanceOf | instanceOf)
    <xsl:apply-templates select="@scope"/>          ->(scope?)
    <xsl:for-each select="assocrl">                 ->(member+)
      <member>
```

```
        <xsl:apply-templates select="@id"/>
        <xsl:apply-templates select="@type" mode="roleSpec"/>
        <topicRef href="#{@href}"/>
      </member>
    </xsl:for-each>
  </association>
</xsl:template>

<xsl:template match="@type|@role">                 ->(instanceOf | instanceOf)
  <xsl:choose>
    <xsl:when test="contains(.,':') or contains(.,'.') or contains(.,'\') or contains(.,'/')">
      <instanceOf><subjectIndicatorRef  href="{.}"/></instanceOf>
    </xsl:when>
    <xsl:otherwise>
      <instanceOf><topicRef href="#{.}"/></instanceOf>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="@scope">                      ->(scope)
  <scope>
    <xsl:call-template name="topicRefSplit">
      <xsl:with-param name="topicRefs" select="."/>
    </xsl:call-template>
  </scope>
</xsl:template>
```

References

[1] Sharon Adler et al. Extensible Stylesheet Language (XSL) version 1.0, October 2001. W3C Recommendation. http://www.w3.org/TR/2001/REC-xsl-20011015/.

[2] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. XML with data values: Typechecking revisited. In *Symposium on Principles of Database Systems*, 2001.

[3] Altova. xmlspy, 2004. http://www.xmlspy.com/.

[4] Amazon.com. Amazon Web Services. http://www.amazon.com/gp/browse.html/102-7542609-2813756?node=3435361, 2002.

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

[6] Jonny Axelsson et al. XHTML 2.0, May 2003. W3C Working Draft. http://www.w3.org/TR/2003/WD-xhtml2-20030506/.

[7] Eric Bae and James Bailey. Codex: An approach for debugging XSLT transformations. In *Proc. 4th International Conference on Web Information Systems Engineering (WISE)*, 2003.

[8] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *Proc. 8th ACM SIGPLAN International Conference on Functional Programming, ICFP '03*, pages 51–63, August 2003.

[9] Scott Boag et al. XQuery 1.0: An XML query language, November 2003. W3C Working Draft. http://www.w3.org/TR/2003/WD-xquery-20031112/.

[10] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2, CSS2 specification, May 1998. W3C Recommendation. http://www.w3.org/TR/1998/REC-CSS2-19980512/.

[11] Ronald Bourret, John Cowan, Ingo Macherius, and Simon St. Laurent. Document definition markup language (DDML) specification, version 1.0, January 1999. W3C Note. http://www.w3.org/TR/NOTE-ddml.

[12] Linda Boyer, Peter Danielsen, Jim Ferrans, Gerald Karam, David Ladd, Bruce Lucas, and Kenneth Rehor. Voice eXtensible Markup Language, version 1.0, May 2000. W3C Note. `http://www.w3.org/TR/voicexml/`.

[13] 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.

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

[15] Tim Bray, Charles Frankston, and Ashok Malhotra. Document content description for XML, July 1998. W3C Note. `http://www.w3.org/TR/NOTE-dcd`.

[16] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML, January 1999. W3C Recommendation. `http://www.w3.org/TR/1999/REC-xml-names-19990114/`.

[17] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.1, February 2004. W3C Recommendation. `http://www.w3.org/TR/2004/REC-xml-names11-20040204/`.

[18] 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/2004/REC-xml-20040204/`.

[19] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1, February 2004. W3C Recommendation. `http://www.w3.org/TR/2004/REC-xml11-20040204/`.

[20] 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, October 2002.

[21] 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, November 2003.

[22] Christian Classics Ethereal Library. Theological markup language (thml), 2004. `http://www.ccel.org/ThML/`.

[23] James Clark. XSL Transformations (XSLT) version 1.0, November 1999. W3C Recommendation.
http://www.w3.org/TR/1999/REC-xslt-19991116.

[24] James Clark. TREX – tree regular expressions for XML, February 2001. http://www.thaiopensource.com/trex/spec.html.

[25] James Clark and Steve DeRose. XML path language version 1.0, November 1999. W3C Recommendation.
http://www.w3.org/TR/1999/REC-xpath-19991116.

[26] James Clark and Makoto Murata. RELAX NG specification, December 2001. OASIS.
http://www.oasis-open.org/committees/relax-ng/.

[27] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1999. Available from
http://www.grappa.univ-lille3.fr/tata/.

[28] John Cowan and Richard Tobin. XML Information Set (second edition), February 2004. W3C Recommendation.
http://www.w3.org/TR/2004/REC-xml-infoset-20040204/.

[29] Andrew Davidson et al. Schema for object-oriented XML 2.0, July 1999. W3C Note. http://www.w3.org/TR/NOTE-SOX/.

[30] Steve DeRose, Eve Maler, and David Orchard. XML linking language (XLink) version 1.0, June 2001. W3C Recommendation.
http://www.w3.org/TR/2001/REC-xlink-20010627/.

[31] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Proc. 8th International World Wide Web Conference, WWW8*, 1999.

[32] Ce Dong and James Bailey. Static analysis of XSLT programs. In *Proc. 15th Australasian Database Conference (ADC)*, January 2004.

[33] Denise Draper et al. XQuery 1.0 and XPath 2.0 formal semantics, February 2004. W3C Working Draft.
http://www.w3.org/TR/xquery-semantics/.

[34] ej-technologies. JProfiler, 2004.
http://www.ej-technologies.com/products/jprofiler/overview.html.

[35] John Ellson et al. Graphviz - open source graph drawing software, 2004. http://www.research.att.com/sw/tools/graphviz/.

[36] J. English. Tcl manual markup language (TMML), 2002. http://tmml.sourceforge.net/index.html.

[37] Jon Ferraiolo et al. Scalable vector graphics (svg) 1.1 specification, January 2003. W3C Recommendation. http://www.w3.org/TR/2003/REC-SVG11-20030114/.

[38] Google. Google Web APIs (beta), 2004. http://www.google.com/apis/.

[39] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. XPointer framework, March 2003. W3C Recommendation. http://www.w3.org/TR/2003/REC-xptr-framework-20030325/.

[40] Haruo Hosoya and Makoto Murata. Boolean operations and inclusion test for attribute-element constraints. In *Proc. 8th International Conference on Implementation and Application of Automata, CIAA '03*, volume 2759 of *LNCS*, pages 201–212. Springer-Verlag, 2003.

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

[42] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proc. 6th ACM SIGPLAN International Conference on Functional Programming, ICFP '00*, pages 11–22, September 2000. Also in SIGPLAN Notices 35(9) (2000).

[43] Keith Isdale, Justin Fletcher, and Igor Zlatkovic. xsldbg, 2004. http://xsldbg.sourceforge.net/.

[44] Masayasu ISHIKAWA. HyperText Markup Language (HTML) home page, 2004. http://www.w3.org/MarkUp/.

[45] Sushant Jain, Ratul Mahajan, and Dan Suciu. Translating XSLT programs to efficient SQL queries. In *Proc. 11th International Conference on World Wide Web*, pages 616–626. ACM Press, 2002.

[46] Rick Jelliffe. The Schematron: An XML structure validation language using patterns in trees, 1999. http://www.ascc.net/xml/resource/schematron/schematron.html.

[47] Jiri Jirat. Zvon XSLTracer. http://www.zvon.org/xxl/XSLTracer/Output/introduction.html.

[48] Michael Kay. XSL Transformations (XSLT) version 2.0, November 2003. W3C Working Draft. http://www.w3.org/TR/2003/WD-xslt20-20031112/.

[49] Michael H. Kay. SAXON DTDGenerator. Available at http://users.breathe.com/mhkay/saxon/dtdgen.html.

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

[51] 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.

[52] Andrew Layman et al. XML-Data, January 1998. W3C Note. http://www.w3.org/TR/1998/NOTE-XML-data/.

[53] Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(3):76–87, 2000.

[54] Håkon Wium Lie and Bert Bos. Cascading Style Sheets, level 1, December 1996. W3C Recommendation. Revised January 1999. http://www.w3.org/TR/1999/REC-CSS1-19990111.

[55] MarrowSoft. Xselerator XSL Editor and Debugger, 2004. http://www.vbxml.com/xselerator/.

[56] Jonathan Marsh and David Orchard. XML Inclusions (XInclude) version 1.0, April 2004. W3C Candidate Recommendation. http://www.w3.org/TR/2004/CR-xinclude-20040413/.

[57] 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.

[58] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66, February 2002. Special Issue on PODS '00, Elsevier.

[59] Mehryar Mohri and Mark-Jan Nederhof. *Robustness in Language and Speech Technology*, chapter 9: Regular Approximation of Context-Free Grammars through Transformation. Kluwer Academic Publishers, 2001.

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

[61] Peter Murray-Rust and Henry Rzepa. The site for chemical markup language, March 2004. http://www.xml-cml.org/.

[62] Frank Neven. Automata theory for XML researchers. *ACM SIGMOD Record*, 31(3), 2002.

[63] OASIS. OASIS DocBook TC.
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=docbook.

[64] Yannis Papakonstantinou and Victor Vianu. DTD Inference for Views of XML Data. In *Proc. 19th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, PODS '00*, pages 35–46, Dallas, Texas, May 2000.

[65] Steven Pemberton et al. XHTML 1.0 The Extensible HyperText Markup Language (second edition), January 2000. W3C Recommendation. Revised 1 August 2002.
http://www.w3.org/TR/2002/REC-xhtml1-20020801/.

[66] Thomas Perst and Helmut Seidl. A type-safe macro system for XML. In *Proc. Extreme Markup Languages*, August 2002.

[67] Benjamin C. Pierce et al. The Xtatic project: Native XML processing for C#, 2004. http://www.cis.upenn.edu/~bcpierce/xtatic/.

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

[69] Dave Raggett. Assertion grammars.
http://www.w3.org/People/Raggett/dtdgen/Docs/, May 1999.

[70] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification, December 1999. W3C Recommendation.
http://www.w3.org/TR/1999/REC-html401-19991224/.

[71] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[72] Chris Stefano. XSLDebugger, 2004.
http://www.vbxml.com/xsldebugger/.

[73] Joint Technical Comittee ISO/IEC JTC1/SC18/WG8, Information technology. *ISO/IEC 10744:1997: Information technology — Hypermedia/Time-based Structuring Language (HyTime).* International Organization for Standardization, 1997. Second Edition.

[74] Joint Technical Comittee ISO/IEC JTC1/SC34, Information technology. *ISO/IEC 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML).* International Organization for Standardization, 1986.

[75] Joint Technical Comittee ISO/IEC JTC1/SC34, Information technology. *ISO/IEC 10179:1996(E): Information technology — Processing languages — Document Style Semantics and Specification Language (DSSSL)*. International Organization for Standardization, 1996.

[76] TEI Consortium. The TEI website, 2004. `http://www.tei-c.org/`.

[77] The Unicode Consortium. *The Unicode Standard, Version 4.0.* Addison Wesley, 2003. `http://www.unicode.org/versions/Unicode4.0.1/`.

[78] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema part 1: Structures, May 2001. W3C Recommendation. `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/`.

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

[80] Eric van der Vlist et al. Examplotron, 2003. `http://examplotron.org/`.

[81] Lionel Villard and Nabil Layaïda. iXSLT: An incremental XSLT transformation processor for XML document manipulation, 2001.

[82] WAP Forum. Wireless Markup Language, version 2.0, September 2001. Wireless Application Protocol Forum. Available from `http://www.wapforum.org/`.

[83] Lauren Wood et al. Document Object Model (DOM) Level 1 specification version 1.0, October 1998. W3C Recommendation. `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`.

[84] World Wide Web Consortium. W3C technical reports and publications, `http://www.w3.org/TR/`.