BRICS RS-03-19 Kirkegaard et al.: Static Analysis of XML Transformations in Java

BRICS

Basic Research in Computer Science

Static Analysis of XML Transformations in Java

Christian Kirkegaard
Anders Møller
Michael I. Schwartzbach

BRICS Report Series RS-03-19

ISSN 0909-0878 May 2003

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

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/03/19/`

Static Analysis of XML Transformations in Java

Christian Kirkegaard, Anders Møller*, and Michael I. Schwartzbach

BRICS†, Department of Computer Science
University of Aarhus, Denmark

Abstract

XML documents generated dynamically by programs are typically represented as text strings or DOM trees. This is a low-level approach for several reasons: 1) Traversing and modifying such structures can be tedious and error prone; 2) Although schema languages, e.g. DTD, allow classes of XML documents to be defined, there are generally no automatic mechanisms for statically checking that a program transforms from one class to another as intended.

We introduce Xact, a high-level approach for Java using XML templates as a first-class data type with operations for manipulating XML values based on XPath. In addition to an efficient runtime representation, the data type permits static type checking using DTD schemas as types. By specifying schemas for the input and output of a program, our algorithm will statically verify that valid input data is always transformed into valid output data and that no errors occur during processing.

1 Introduction

Extensible Markup Language, XML [10], has since its introduction in 1998 gained considerable interest from industry and now plays an important role in the exchange of a wide variety of data on the Web. Although XML, technically, is merely a linear syntax for ordered labeled tree structures, it has proven useful as a notation for structuring information in general.

The syntax of an XML-based language is specified using a vocabulary of elements and attributes together with rules for constraining their use. There exists a variety of schema languages, such as DTD [10], XML Schema [46], or DSD2 [34], allowing the syntax to be formalized. An XML document is *valid* relative to a given schema if all the syntactic requirements specified by the

*Corresponding author. Email: `amoeller@brics.dk`

†This work is supported by Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation. Anders Møller is supported by the Carlsberg Foundation contract number ANS-1069/20.

schema are satisfied in the document. The *language* $\mathcal{L}(S)$ of a schema S is the set of XML documents that are valid relative to S.

A popular XML-based language is XHTML [38], the "XMLized" variant of HTML. The XHTML language is widely used in *interactive* Web services where the clients are human beings that use browsers to interact with the servers. A recent trend is to move from interactive Web services towards *application-to-application* Web services, where the clients are not humans with browsers but general programs. This calls for specialized XML-based languages to mediate communication between clients and servers. As an example, Amazon.com now provides an XML interface [1] that allows other programs to search for product information. These other programs may combine that information with data from other sources, extract relevant parts and, for example, transform the results into other XML documents to interact with yet another group of programs.

From this development, it is clear that XML already plays a central role in representation of information on the Web and that *transformation* of XML data is becoming a key aspect of Web service programming.

Existing general-purpose programming languages do not provide any special support for XML transformations. With these languages, the programmer may choose to model XML data either 1) as text strings, or 2) as DOM [2] tree structures (or variants of that, such as JDOM [26]). The first approach is often used for languages as XHTML where documents are being constructed but rarely deconstructed, whereas the second is more used for languages and transformation that involve both construction and deconstruction of documents. We shall argue that both approaches are low-level in the sense that they are often error-prone and tedious to use.

Our ultimate goal is to integrate XML into general-purpose programming languages, in particular Java, to support more high-level definitions of XML transformations and thereby make development of Web services easier and safer.

We wish to incorporate XML data as first-class values in Java. Since an XML schema defines a class of XML documents, it is natural to view schemas as *types* alongside the standard types such as integers and strings. An XML transformation is defined by a program that as input takes one or more XML documents $x_1^{in}, \ldots, x_n^{in}$ and as output produces a new XML document x^{out}. In the same way the notion of types is normally used in programming for structuring the code and detecting programming errors at an early stage, the program may assume that each input document x_i^{in} is valid relative to some input schema S_i^{in}, and it is intended that the output document x^{out} is always valid relative to some output schema S^{out}. In this article we wish to:

1. incorporate XML into Java with a family of basic but high-level operations for defining transformations, and
2. provide *static type checking*, that is, for the program, verify at compile-time that $x^{out} \in \mathcal{L}(S^{out})$ given that $x_i^{in} \in \mathcal{L}(S_i^{in})$ for each i.

In comparison, the existing approaches of using text strings or DOM trees do not support static type checking.

We work in the context of JWIG [16, 13], an extension of Java that, among other features, provides a mechanism for construction of XML documents using *XML templates* and *plug operations*, which we briefly recapitulate in Section 3. Our previous results included a static analysis for checking that the constructed documents are always valid relative to a given DSD2 schema. However, the mechanism only supported *construction* of XML documents, not *deconstruction*. This has shown to be sufficient for interactive Web services that dynamically create XHTML documents, but, as explained earlier, application-to-application Web services require general XML transformations, which also includes deconstruction. Furthermore, the previous results were obtained under the assumption that XML documents are built from a set of constant XML templates. This is also a valid assumption for interactive Web services, but not for application-to-application Web services, where the constituents of the result of an XML transformation are often input from other Web services. In the present article we generalize the previous results to general XML transformations that also involve deconstruction and importing of XML templates.

Contributions

Our contributions in this article are the following:

- A survey of existing techniques for defining XML transformations;
- a novel data type with high-level operations for defining XML transformations in Java;
- a static analysis technique based on a notion of summary graphs;
- an algorithm for symbolic evaluation of XPath expressions [19] on summary graphs;
- an algorithm for converting DTD schemas into summary graphs; and
- experimental evidence that the approach is practically feasible.

Preliminary results were described in [14]. In a separate paper [15], we show that our data type permits an efficient runtime representation. Although we focus on Java, our ideas can be applied to other general-purpose high-level programming languages since we do not depend on any Java-specific language constructs.

Overview

We first, in Section 2, describe related work on language support for XML transformations and motivate the need for new solutions. Section 3 explains our approach that involves DTD and XPath. The operations can be performed efficiently with a suitable runtime representation, which we mention briefly and describe in detail in a separate paper. In Section 4, we describe *summary graphs*, a formalism that provides the foundation for the static program analysis, which we describe in Section 5. This analysis encompasses techniques for symbolically evaluating XPath expressions on summary graphs and converting DTD schemas into summary graphs. Our prototype implementation and a number of benchmark tests of the analyzer are described in Section 6.

2 Related Work

There exists a wide range of approaches for defining XML transformations, originating from database, hypertext, and programming language communities. These approaches are in the following divided into techniques for *general-purpose* programming languages and for tailor-made *domain-specific* languages. A general introduction to the XML type checking problem is given in [41].

XML data may be manipulated in several ways that are not all supported equally well by every approach. In many actual XML transformations, the input and output languages are different, i.e., described by different schemas. However, often these languages are the same, for example if the transformation consists of sorting a list of entries in a table but leaving the rest of the document unmodified. Such transformations are often described more conveniently as in situ modifications than as functions from input to output. Also, many programs involving XML build documents from non-XML sources, extract information from XML without producing XML output, or they interact with other systems during the processing. Developing good support for XML in programming also requires consideration of these pragmatic issues.

Techniques for general-purpose languages

The approaches of representing XML data as strings or DOM trees, as mentioned in the introduction, fit into the category of techniques for general-purpose languages. Building XML documents by concatenating string fragments is commonly used in the presentation layer of interactive Web services, for example with Servlets [43]. This primitive approach does not assist the programmer in avoiding mismatching tags or improper escaping of special characters, and it does not support deconstruction of documents.

Presently, there are XML libraries with parsers and DOM-like functionality for all major (and also many less widely used) programming languages. Examples for Java include JDOM [26], TrAX [7], and JAXP [42]. Such libraries view XML data as tree structures and provide operations for local traversal and manipulation. This is a powerful approach that permits the full underlying programming language to be involved in the XML processing. Wellformedness of the involved XML data comes for free when working on the tree level. However, it is still a low-level approach for a number of reasons: 1) Traversing or modifying a DOM tree is expressed via primitive operations, for example taking a single step in the tree from an element to its first child element. More complex operations therefore tend to require relatively much code, compared to e.g. XSLT, which is described below. 2) There is no tool support for analyzing the programs at compile-time to verify that transformation output is guaranteed to be valid at runtime or that the transformations succeed without runtime errors. XML is regarded as one homogeneous type without considering schemas. The processing is completely independent from the schema information, so, for example, a schema may contain the information that A elements cannot occur as children of B elements, but failed attempts to select an A child element of a B element in a program will not be detected until runtime.

SAX [11] is event-based rather than tree-based. This approach is suitable for streaming processing of large documents, but static validity is not considered.

To attack the problem of statically guaranteeing validity of the transformation output, a number of systems attempt to model XML transformation using pre-existing type systems in general-purpose programming languages. Examples based on functional languages are HaXml [48] and WASH/CGI [45], both embedding DTD into Haskell. In contrast to HaXml, WASH/CGI does not support deconstruction of XML values. In return, WASH/CGI allows the use of generic combinators, which the type-safe approach in HaXml does not.

With this approach, type checking of XML transformations comes for free via the type system in the host language. However, these type systems are usually not strong enough to capture all requirements specified in a schema without sacrificing soundness, performance, or flexibility [25], even with a simple schema language as DTD. Another problem is that type errors are reported at the level of the underlying host language, which can make them difficult to understand for the programmer.

Other systems are targeted at object-oriented languages, typically Java. Castor [22] and the more recent JAXB [44] are XML data binding frameworks for Java. From a schema written in certain subsets of XML Schema they can generate a collection of Java classes representing an object model of the corresponding XML documents. XML data may then be processed as Java objects at a higher abstraction level than e.g. JDOM. Methods for marshalling and unmarshalling are automatically generated, and the mapping between XML and Java can be controlled by specifying explicit bindings. Relaxer [23] is a similar tool but for the RELAX schema language. For all three systems, there is no static guarantee that a constructed document will satisfy all the requirements of the given schema.

The SNAQue tool [40] provides a variant of data binding that does not take schemas into account. From an XML document and a programming language type, it extracts a program value. Projector [20] is a related extension of JavaScript mixing typed and untyped programming.

The approach described in [31] contains a data binding system for languages with powerful types with streams, tuples, and unions, which allow schemas to be encoded with high precision. A type checking algorithm is currently being implemented but is yet unpublished. Many other data binding tools are described in [8].

Domain-specific languages

Domain-specific languages (DSLs) are tailor-made for specialized classes of tasks, such as XML transformation. Although the formal expressive power of these language of course does not exceed that of general-purpose languages, the advantages of DSLs are generally considered to be 1) high levels of abstraction with language constructs and customized syntax that closely match the concepts in the problem domain, and 2) specialized analyses for reasoning about the behavior of programs.

The predominant DSL for XML transformation is XSLT [18], a declarative language based on pattern matching and template instantiation. Although designed primarily for hypertext stylesheet applications, it is more widely applicable, for example, for simple database operations. XSLT uses XPath for pointing and pattern matching. Schemas for the input and output languages are ignored by XSLT processors, so no type checking is performed.

Although DSLs for XML transformation certainly do have a raison d'être, many have difficulties with the kinds of transformation mentioned earlier that involve non-XML values or need to interact with other systems. XSLT is extensible, but only in the sense that individual implementors may add their own extra functionality.

XQuery [6] can be viewed as a generalization of SQL to the richer data model of XML. It is a functional language with optional types using a considerable subset of XML Schema as basis for its type system [21], which supports static type inference and checking. Although still at working draft level with many open issues, XQuery is an ambitious project and receives much attention.

XDuce [25] is a simplistic functional language based on regular expression types, which are a natural generalization of DTD schemas, and a corresponding mechanism for pattern matching. It supports a local form of type inference where types are specified explicitly for function arguments but inferred for pattern matching. In its current version, XDuce does not have higher-order functions or parametric polymorphism, and the type system does not model element attributes or unordered data. ℂDuce [3] extends XDuce into a full programming language and adds higher-order functions and other language features. The ideas from XDuce, which have also influenced the design of the XQuery type system, are currently being integrated into C# in the Xtatic project with similar goals as ours [24].

XMλ [32] is a functional language related to HaXml and WASH/CGI. Its type system uses a notion of type-indexed rows to model DTD. Whereas subtyping is an essential aspect in XDuce, XMλ is based on parametric polymorphism. Apparently, no implementation of XMλ is available.

The language *fxt* [4] is closely related to XSLT but uses a strictly top-down processing model and a clean pattern matching mechanism that corresponds to regular languages. Another attempt to redesign XSLT is SXSLT [29], based on Scheme. Both *fxt* and SXSLT focus on language design and, as XSLT, do not provide type checking.

The type checking problem has been studied at a more theoretical level for *k-pebble tree transducers* [33], a framework for modeling decidable tree transformations in, for example, a predecessor of XQuery and a fragment of XSLT. A less expressive formalism for top-down transformations is investigated in [30], and another related approach is proposed in [47] for type checking a subset of XSLT using tree automata.

In [39], a simple XML transformation system based on macro expansion is described, and it is shown that exact type checking with DTD is decidable for this system. The query language *loto-ql* permits inference of output schemas from input schemas using a generalization of DTD to context-free languages [37].

Finally, we mention the recent XOBE language [28], which is closely related to our approach. XOBE is also an extension of Java, it has a notion of XML templates resembling that of JWIG, and it too uses XPath to select parts of XML trees. XOBE uses a type system based on regular hedge grammars, whereas we rely on dataflow analysis using summary graphs to obtain static guarantees. However, there are a number of more essential differences: XML trees in XOBE can only be constructed bottom-up. In contrast, the template mechanism in JWIG is higher-order in the sense that templates can contain named gaps that can be filled in any order, possibly with templates containing other gaps. Also, a template in XOBE always has exactly one root element; JWIG templates allow sequences of elements and character data at the top level. XOBE requires all variables to be explicitly typed with element names, unlike our approach. Lists of mixed elements cannot be built dynamically in XOBE since it only supports lists of elements of the same type. Finally, our *gapify* construct, which we describe later, has no counterpart in XOBE. These issues make our mechanism considerably more flexible in practice.

Our approach

We present a technique, XACT, that combines 1) a full integration of XML values and highly flexible operations for XML transformation into an existing high-level language, and 2) static guarantees of type safety of the transformations.

We choose to build on Java since this language is already widely used in development of Web services. Using a general-purpose language allows mixing XML manipulations with other functionality, for example, accessing data bases or communicating on the Internet. Our starting point is the XML template mechanism in JWIG. We use XPath for selecting fragments of XML values. XPath has already proven useful for this purpose in, e.g., XSLT and XQuery.

Our approach to providing static guarantees is based on dataflow analysis rather than type systems. Dataflow analysis works on control-flow graphs, which allows flow sensitivity, whereas type systems typically work on abstract syntax trees. Our analysis is reminiscent of type inference since variable declarations do not have explicit types.

By building on an imperative language, our mechanism is operational and in that respect closer to, for example, JDOM, than to a declarative language as XQuery. However, an important design choice is that our data type for XML templates is *immutable* [5]. There are several reasons for this choice: As in pure functional languages, having no side-effects often permits a cleaner programming style. For example, there is no need for explicit copying of values, thread safety comes for free, and the use of value factories is allowed. Furthermore, since side-effects can be difficult to control, having immutable data avoids a significant class of programming errors. Finally, the crucial point in our situation is that immutability is a necessity for development of a feasible program analysis. It would not be possible to transfer our program analysis techniques to a mutable data type as, e.g., JDOM.

3 XML Operations using DTD and XPath

We represent XML values as *XML templates* in the style of JWIG [16]. An XML template is a wellformed XML fragment that may contain named *gaps* where other templates or strings may be inserted. The gaps may appear in place of elements or attribute values. In JWIG, this has proven to constitute an intuitive and flexible mechanism for XML document construction.

Formally, XML templates are derived by *xml* in the following grammar:

xml	:	*str*	(character data)
	|	<*name atts*> *xml* </*name*>	(element)
	|	<[*g*]>	(template gap)
	|	*xml xml*	
atts	:	*name*="*str*"	(attribute constant)
	|	*name*=[*g*]	(attribute gap)
	|	*atts atts*	
	|	ε	

Here, *str* denotes an arbitrary Unicode string, *name* is an identifier, and g is a gap name. Actual XML values must of course be further constrained to be wellformed according to the XML 1.0 specification [10]. Moreover, in this description we abstract away all inlined DTD information, comments, and processing instructions. Compared to the description in [16], we here omit code gaps.

In this article, we extend the JWIG mechanism with operations for deconstructing and importing XML data. These operations are based on DTD and XPath, which we briefly describe in the following to explain the terminology that we use.

DTD

The DTD formalism is a simple schema language for XML and is described in the XML specification [10]. A DTD schema is a grammar for a class of XML documents defining for each element the required and permitted child elements and attributes. The *contents* of an element is the sequence of its immediate children. It is specified using a restricted form of regular expression over element names and `#PCDATA`, which refers to arbitrary character data. Attributes can be declared as required or optional for a given element, and their values can be constrained to finite collections of fixed strings. We ignore the special attribute types `ID`, `IDREF`, `ENTITY`, etc.

The following example is a DTD schema for collections of recipes:

```
<!DOCTYPE collection [
  <!ELEMENT collection (title,recipe*)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT recipe (title,ingredient*,preparation)>
  <!ELEMENT ingredient (ingredient*,preparation)?>
  <!ATTLIST ingredient name CDATA #REQUIRED
                       amount CDATA #IMPLIED
                       unit CDATA #IMPLIED>
  <!ELEMENT preparation (step*)>
  <!ELEMENT step (#PCDATA)>
]>
```

This data model support both *simple* ingredients, consisting of a `name` and possibly an `amount` and a `unit`, and *composite* ingredients, which are described recursively by sub-recipes. The following small example is valid according to the schema:

```
<collection>
  <title>Tiny Example</title>
  <recipe>
    <title>Cucumber Sandwich</title>
    <ingredient name="cucumber"/>
    <ingredient name="crustless bread">
      <ingredient name="bread" amount="1" unit="loaf"/>
      <preparation>
        <step>Cut one slice of the bread.</step>
        <step>Cut the crust from the slice.</step>
      </preparation>
    </ingredient>
    <ingredient name="mayo" amount="2" unit="dollop"/>
    <preparation>
      <step>Spread the mayo on the bread.</step>
      <step>Slice the cucumber.</step>
      <step>Place cucumber slices on the bread.</step>
    </preparation>
  </recipe>
</collection>
```

The JWIG validity analysis described in [16] uses a more powerful schema language, DSD2 [34], which is capable of capturing more complex syntactic requirements than DTD. The main reason for using DTD here is that our generalization of the XML cast operation, as explained in the following sections, requires translation from schemas into our summary graphs, which can be done straightforwardly and precisely for DTD.

XPath

XPath [19] is a simple but versatile DSL for addressing elements, attribute values, and character data—generally called *nodes*—in XML documents. It has proven powerful as a sub-language, for example in XSLT, for locating document fragments and as a pattern matching mechanism.

An XPath *expression* can, relative to an evaluation context, evaluate to a boolean, a number, a string, or a set of nodes. A node set expression is called a *location path* and consists of a sequence of *location steps*, each having three parts: 1) an *axis*, for example `child` or `following-sibling`, which selects a set of nodes relative to the context node, 2) a *node test*, which filters the selected nodes by considering their type or name, and 3) a number of *predicates*, which are boolean expressions that perform a further, potentially more complex, filtration. Thus, the result of evaluating a location step on a specific node is a set of nodes. A whole location path is evaluated compositionally left-to-right. As an example, the following expression selects all `amount` attributes in `ingredient` elements that have a `name="salt"` attribute and occur within `recipe` elements that have a `title` child with contents `soup`:

```
child::recipe[string(child::title/child::text())="soup"]/
descendant-or-self::ingredient[string(attribute::name)="salt"]/
attribute::amount
```

where we assume that the initial context node is a `collection` element. The `string()` function extracts the string value of a node.

In our application of XPath, we restrict ourselves to the `child`, `descendant-or-self`, and `attribute` axes. This means that all evaluation is then top-down, which is sufficient for all the transformations we mention in Section 6 and simplifies both the runtime system and the analyzer. A similar approach is taken in the *fxt* language [4]. Conveniently, XPath offers some syntactic sugar for these axes: `child` is the default axis, `descendant-or-self` may be written as `//`, and `attribute` may be written as `@`. The example above may then be abbreviated as follows:

```
recipe[title/text()="soup"]//ingredient[@name="salt"]/@amount
```

where we also use an implicit coercion rule converting nodes to their textual contents.

Basic XML Operations

The class `XML`, which represents XML templates, allows several operations that are shown in Figure 1.

The `const` operation constructs an XML template from the syntax generated by the *xml* nonterminal in the previously described grammar; the `toString` operation translates in the opposite direction. The argument to `const` must be a constant.

```
static XML const(String s)
   – creates an XML template from a constant string
String toString()
   – converts this XML template into its textual representation
XML plug(Gap g, XML x)
   – inserts a copy of x into every occurrence of g in a copy of
     this template
XML plug(Gap g, String s)
   – as the previous, but for a string
XML plug(Gap g, XML[] xs)
   – inserts the template items in xs into the g gaps in a copy of
     this template
XML plug(Gap g, String[] ss)
   – as the previous, but for a string array
XML close()
   – removes all open template gaps and all attributes with open gaps
XML[] select(XPath p)
   – returns all sub-templates selected by p
XML[] cut(XPath p)
   – as select, but only returns maximal disjoint sub-templates
XML gapify(XPath p, Gap g)
   – copies this template and converts all sub-templates selected by
     p into g gaps
static XML smash(XML[] xs)
   – merges the templates in xs into one template by concatenating them
String text()
   – returns the concatenation of all character data at the top level of
     this template
XML cast(DTD d)
   – throws a runtime exception if this template is invalid relative to d
static XML get(String s, DTD d)
   – converts s into a template and checks validity relative to d
XML analyze(DTD d)
   – instructs the analyzer to verify that this template is valid
     relative to d
```

Figure 1: Methods in the XML interface for performing basic XML template operations.

The `plug` operation is defined in four variants accepting strings, templates, or arrays of these. In the array versions, all occurrences of the named gap are in document order plugged with the values occurring in the array. If the lengths do not match, then superfluous array values are ignored and remaining gaps are plugged with the empty template. For the case where an element contains multiple attribute gaps, these are ordered lexicographically by attribute name. In the non-array version, all occurrences of the named gap are plugged with the given value. Attempts to plug templates into attribute gaps will result in runtime errors. A gap that has not been plugged is said to be *open*. The `close` operation closes all gaps by removing template gaps and for each attribute gaps, the entire attribute is removed.

The `select`, `cut`, and `gapify` operations first find the node set indicated by the XPath expression using an implicit root node as evaluation context. In `select` and `cut`, the subtrees rooted by nodes in this set are copied in document order to form the resulting template array. Attribute gaps in the node set are ignored, and for normal attributes, their values are converted into character data. In `gapify`, the selected nodes and their sub-trees are each replaced by a gap with the given name. For `cut` and `gapify`, if one selected node is an ancestor of another, then only the ancestor is considered. This means that `cut` always returns disjoint subtrees in contrast to the variant `select`, which may return overlapping subtrees.

The `smash` operation concatenates an array of templates into a single template. The `text` operation concatenates all character data occurring at the top level.

The `cast` operation checks that the template is valid according to the given DTD schema and throws an exception otherwise. The `get` operation converts a non-constant string into a template that is then validated according to the given DTD. The `analyze` operation has no effect at runtime but instructs the analyzer to verify that the template is valid relative to the given DTD.

All arguments of types `Gap`, `XPath`, and `DTD` are required to be constant.

Note that, e.g., all JDOM operations trivially are special cases of these operations – except that our data type is immutable, as explained earlier.

An XML transformation typically has the following form:

```
String transform(String s) {
  XML x = XML.get(s, DTD.make("http://.../input.dtd"));
  ...
  return x.analyze(DTD.make("http://.../output.dtd"))
          .close().toString();
}
```

where input and output XML is represented textually.

The program analysis described later will at compile time check that 1) each `analyze` operation is valid in the sense that the given template at runtime is guaranteed to be valid relative to the DTD schema, and 2) each `plug` operation always succeeds, that is, the gap is guaranteed to be present and templates are never plugged into attribute gaps. Furthermore, if the analysis detects that an

XPath expressions in a `select`, `cut`, or `gapify` operations will never select any nodes, a warning is issued.

Syntactic Sugar

The JWIG language permits some syntactic sugar on top of the basic operations. First, we allow special syntax for template constants, which may be written in `[[...]]` without the otherwise mandatory escape characters. Similarly, arguments of types `Gap` and `XPath` may be written directly without explicit calls to constructors. Also, if f is a local method accepting exactly one argument of type `XML` and whose result is of type `XML`, then `[]`f abbreviates a new local method that accepts and returns arguments of type `XML[]` and applies f to each array entry. Finally, we allow the simple abbreviations:

```
x.roots() ≡ x.select(*)
x.attribute(a) ≡ smash(x.select(@a)).text()
x.chardata() ≡ smash(x.select(text())).text()
x.has(p) ≡ x.select(p).length>0
x.size() ≡ x.roots().length
x.delete(p) ≡ x.gapify(p,g).plug(g,"")
x.apply(p,f) ≡ x.gapify(p,g).plug(g,[]f(x.cut(p)))
```

The operations satisfy some simple equations, which may further elucidate their semantics:

```
x.toString() = const(x.toString()).toString()
x = smash(x.roots())
x = x.gapify(p,g).plug(g,x.cut(p))
```

Consider a method `upperTitle` that creates a copy of a recipe collection in which all titles are raised to upper case. The following sugared syntax:

```
XML toUpper(XML x) {
  return [[<title><[t]></title>]].plug(t, x.chardata().toUpperCase());
}
XML upperTitle(XML x) {
  return x.apply(//title, []toUpper);
}
```

then abbreviates the more cumbersome basic syntax:

```
XML toUpper(XML x) {
  return XML.const("<title><[t]></title>")
             .plug(new Gap("t"),
                   XML.smash(x.select("text()")).text().toUpperCase());
}
XML toUpperArray(XML[] x) {
  XML[] y = new XML[x.length];
  for (int i=0; i<x.length; i++) y[i]=toUpper(x[i]);
```

```
  return y;
}
XML upperTitle(XML x) {
  return x.gapify("//title", new Gap("n"))
          .plug(new Gap("n"),
                toUpperArray(x.cut("//title")));
}
```

These syntactic extension to Java can be implemented using the Metafront tool [9].

The following complete example implements the recursive TREE Q6 query from the XQuery use cases [12]:

```
XML summary(XML[] x) {
  XML y[] = new XML[x.length];
  for (int i=0; i<x.length; i++)
    y[i] = [[<section id=[id] difficulty=[diff]>
              <title><[t]></title>
              <figcount><[f]></figcount>
              <[s]>
            </section>]]
      .plug(id, x[i].attribute(id))
      .plug(diff, x[i].attribute(difficulty))
      .plug(t, x[i].select(section/title))
      .plug(f, x[i].select(section/figure).length)
      .plug(s, summary(x[i].select(section/section));
  return XML.smash(y);
}
String Q6(String s) {
  XML x = XML.get(s, DTD.make("book.dtd"));
  return [[<toc><[t]></toc>]]
         .plug(t, summary(x.select(book/section)))
         .analyze(DTD.make("Q6.dtd")).toString();
}
```

The structure of this code is similar to the XQuery version.

Runtime Representation

We show in a separate paper [15] that our data type for XML templates permits an efficient runtime representation, despite being immutable. We use a lazy non-copying data structure in which operations are merely noted to have happened until their effects are required to be observed. We obtain nearly optimal asymptotic complexities of the basic operations, since `plug` and individual moves from a parent node to its first child and from a node to its next sibling happen in amortized almost constant time. The `toString` operation is performed in linear time in the size of the resulting string. The complexity of `select`, `cut`, and `gapify` is bounded by the evaluation time for the associated XPath expression. All this assumes that we avoid a pathological case where templates containing only gaps are nested to an unbounded depth. We expect that a tuned implementation will compare favorably with the runtime performances of JDOM, XSLT, or XDuce. The `analyze` operation has no effect at runtime. The `cast` and `get` operations perform a linear time DTD validation.

4 Summary Graphs

To obtain static guarantees, we apply the standard dataflow analysis framework [36, 27]. This involves three steps: 1) obtaining an abstract *control-flow graph* for the given program; 2) defining a *lattice* modeling the abstract data that the analysis manipulates; and 3) describing all kinds of operations in the control-flow graph in terms of *transfer functions* that operate on the lattice values.

The construction of control-flow graphs from Java programs is described in detail in [16]. We use a different family of statements here, but the overall approach is the same and we do not describe it further—however, we note that arrays are modeled by merging their entries and using weak updating. Our lattice is a variant of the *summary graph* lattice defined in [16] – we here use a notion of *normalized* summary graphs, as defined below. The transfer functions are described in Section 5.

Given a program and all DTD schemas it refers to in `cast` and `get` operations, we fix a number of sets and functions that are used by all summary graphs that occur during the analysis: The sets E, A, and G contain the element names, attribute names, and gap names, respectively, that occur in the program and in the schemas. Let $N_{\mathcal{E}}$, $N_{\mathcal{A}}$, $N_{\mathcal{T}}$, and $N_{\mathcal{C}}$ be finite disjoint sets of element, attribute, template, and chardata nodes, respectively.

- $N_{\mathcal{E}}$ contains a node for each occurrence of an element in a template constant in the program and one for each element description in the schemas. The function $name : N_{\mathcal{E}} \to E$ returns the corresponding element name.

- $N_{\mathcal{A}}$ contains a node for each occurrence of an attribute in a template constant and one for each attribute description in the schemas. The function $name : N_{\mathcal{A}} \to A$ returns the corresponding attribute name. Each element node is associated a set of attribute nodes, $attr : N_{\mathcal{E}} \to 2^{N_{\mathcal{A}}}$ corresponding to the element attributes.

- $N_{\mathcal{T}}$ contains a node for each node in $N_{\mathcal{E}}$, one for each template constant, one for each occurrence of `select`, `cut`, `smash`, or `gapify`, and one for each sub-expression of the content model descriptors in the schemas. Each element node is associated a template node, $contents : N_{\mathcal{E}} \to N_{\mathcal{T}}$, corresponding to the element contents. Each template node has a sequence of gaps, $gaps : N_{\mathcal{T}} \to G^*$, which we define in Section 5.

- $N_{\mathcal{C}}$ contains a node for each maximal chardata sequence in a template constant and one for each occurrence of `plug`, `select`, `cut`, and `#PCDATA`.

The set of all nodes is $N = N_\mathcal{E} \cup N_\mathcal{A} \cup N_\mathcal{T} \cup N_\mathcal{C}$. Note that two elements that have identical names but occur in distinct template constants are modeled by distinct element nodes. This ensures an important form of polyvariance in the analysis.

A *(normalized) summary graph SG* is then a structure:

$$SG = (R, T, S, P)$$

where:

$R \subseteq N_\mathcal{E} \cup N_\mathcal{T}$ is a set of *root nodes*,
$T \subseteq N_\mathcal{T} \times G \times (N_\mathcal{T} \cup N_\mathcal{E} \cup N_\mathcal{C})$ is a set of *template edges*,
$S : N_\mathcal{C} \cup N_\mathcal{A} \to REG$ is a *string edge* map, and
$P : G \to 2^{N_\mathcal{A} \cup N_\mathcal{T}} \times 2^{N_\mathcal{A} \cup N_\mathcal{T}} \times \Gamma \times \Gamma$ is a *gap presence* map.

Here $\Gamma = 2^{\{\mathsf{OPEN},\mathsf{CLOSED}\}}$ is the *gap presence lattice* whose ordering is set inclusion. The set REG is a finite family of regular languages over the Unicode alphabet obtained by a separate analysis of string operations [17].

Intuitively, the language $\mathcal{L}(SG)$ of a summary graph SG is the set of XML templates that can be obtained by unfolding it, starting from a root node and plugging elements, templates, and strings into gaps according to the edges. A template edge $(n_1, g, n_2) \in T$ informally means that n_2 may be plugged into the g gaps in n_1, and a string edge $S(n) = L$ means that every string in L may be plugged into the gap in n.

We need the gap presence map to determine where edges should be added when modeling `plug` operations, to model the removal of gaps with the `close` operation, to detect when `plug` operations may fail because the specified gaps have already been closed, and to model and check XPath evaluations. Given that $P(g) = (p_1, p_2, p_3, p_4)$, let $open(P(g)) = p_1$, $removed(P(g)) = p_2$, $tgaps(P(g)) = p_3$, $agaps(P(g)) = p_4$. Informally, the *open* and *removed* components specify which nodes may contain open or removed g gaps, and *tgaps* and *agaps* describe the presence of template gaps and attribute gaps, respectively. The value OPEN means that the gaps may be present, and CLOSED means that they may be absent.

As a example, we can define a summary graph whose language is the set of `ul` lists with zero or more `li` items that each contain a string from some language L: Assume that the fixed structure is given by $N_\mathcal{E} = \{1, 4\}$, $N_\mathcal{A} = \emptyset$, $N_\mathcal{T} = \{2, 3, 5\}$, $N_\mathcal{C} = \{6\}$, $contents(1) = 2$, $contents(4) = 5$, $attr(1) = attr(4) = \emptyset$, $name(1) = \texttt{ul}$, $name(4) = \texttt{li}$, $gaps(2) = \textit{\texttt{items}}$, $gaps(3) = \textit{\texttt{g}} \cdot \textit{\texttt{items}}$, and $gaps(5) = \textit{\texttt{text}}$. Now define the summary graph (R, T, S, P):

$R = \{1\}$
$T = \{(2, \textit{\texttt{items}}, 3), (3, \textit{\texttt{items}}, 3), (3, \textit{\texttt{g}}, 4), (5, \textit{\texttt{text}}, 6)\}$
$S(6) = L$
$P(\textit{\texttt{text}}) = P(\textit{\texttt{g}}) = (\emptyset, \emptyset, \{\mathsf{CLOSED}\}, \{\mathsf{CLOSED}\})$
$P(\textit{\texttt{items}}) = (\{2, 3\}, \emptyset, \{\mathsf{OPEN}\}, \{\mathsf{CLOSED}\})$

This can be illustrated as follows:

$$\frac{n \in E_{\mathcal{E}} \quad SG \vdash \mathit{contents}(n) \Rightarrow d \quad \mathit{name}(n) = e \quad \mathit{attr}(n) = \{a_1, \ldots, a_k\} \quad SG \vdash a_i \Rightarrow b_i}{SG \vdash n \Rightarrow \texttt{<}e\ b_1 \ldots b_k\texttt{>}\ d\ \texttt{</}e\texttt{>}}$$

$$\frac{n \in E_{\mathcal{C}} \quad s \in S(n)}{SG \vdash n \Rightarrow s} \qquad \frac{n \in E_{\mathcal{A}} \quad \mathit{name}(n) = a \quad s \in S(n)}{SG \vdash n \Rightarrow a\texttt{="}s\texttt{"}}$$

$$\frac{n \in E_{\mathcal{A}} \quad \mathit{name}(n) = a \quad n \in \mathit{open}(P(g))}{SG \vdash n \Rightarrow a\texttt{=[}g\texttt{]}}$$

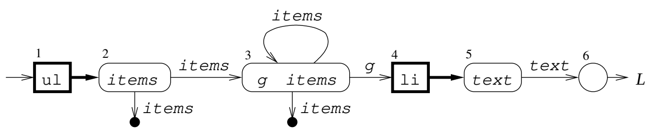
$$\frac{n \in E_{\mathcal{A}} \quad n \in \mathit{removed}(P(g))}{SG \vdash n \Rightarrow \epsilon}$$

$$\frac{n \in E_{\mathcal{T}} \quad \mathit{gaps}(n) = g_1 \ldots g_k \quad SG, g_i \vdash d_i}{SG \vdash n \Rightarrow d_1 \ldots d_k}$$

$$\frac{(n, g, m) \in T \quad SG \vdash m \Rightarrow d}{SG, g \vdash n \Rightarrow d}$$

$$\frac{n \in \mathit{open}(P(g))}{SG, g \vdash n \Rightarrow \texttt{<[}g\texttt{]>}} \qquad \frac{n \in \mathit{removed}(P(g))}{SG, g \vdash n \Rightarrow \epsilon}$$

Figure 2: Inference rules for unfolding of summary graphs.



The boxes represent element nodes, rounded boxes are template nodes, the circle is a chardata node, and the dots represent potentially open template gaps.

The family of summary graph structures forms a lattice using a pointwise subset ordering. For a fixed program, the lattice has finite height.

The unfolding of summary graphs can be formalized as:

$$\mathit{unfold}(SG) = \{d \mid \exists r \in R : SG \vdash r \Rightarrow d\}$$

where the *unfolding relation*, $\Rightarrow$, is defined by induction in the structure of the summary graph according to Figure 2. We define the language of a summary graph as:

$$\mathcal{L}(SG) = \{\mathit{close}(d) \mid d \in \mathit{unfold}(SG)\}$$

where $\mathit{close}(d)$ removes all occurrences of template gaps and attribute gaps.

Compared with the definition of summary graphs in [16], a node now corresponds to at most one chardata sequence, element, or attribute—corresponding to the possible targets of XPath evaluation. Furthermore, we have added the *removed* component of the gap presence map to model the `close` operation.

5 Modeling XML Operations on Summary Graphs

Our dataflow analysis associates a summary graph SG with every `XML` variable and expression at every program point. The analysis is conservative meaning that $\mathit{unfold}(SG)$ contains all XML templates that may occur at that point at runtime.

The essence of the dataflow analysis is the definition of transfer functions for the XML operations. Let Δ denote an environment that maps each `XML` variable to a summary graph. The transfer function for an assignment x=exp is:

$$\Delta \mapsto \Delta[x \mapsto \widehat{\Delta}(\mathit{exp})]$$

and for all other statements, it is the identity function. The function $\widehat{\Delta}$ extends Δ to `XML` expressions according to the expression kind:

`const`: We show below how to construct a summary graph SG_{xml} for a given template constant `[[`xml`]]`.

`plug`: All four variants of `plug` operations are modeled essentially as in [16], so we omit the formal definition here. A template plug invocation exp_1`.plug(`g, exp_2`)` is modeled by adding template edges from nodes with open g gaps in $\widehat{\Delta}(\mathit{exp}_1)$ to roots in $\widehat{\Delta}(\mathit{exp}_2)$. A string plug is modeled by collecting the possible strings into the associated chardata node. The new *removed* component of the gap presence map of the result is defined in the same way as the *open* component.

`close`: To model the removal of gaps, we define $\widehat{\Delta}(\mathit{exp}\texttt{.close()}) = (R, T, S, \lambda h.(\emptyset, \mathit{removed}(P(h)) \cup \mathit{open}(P(h)), \{\mathsf{CLOSED}\}, \{\mathsf{CLOSED}\}))$ where $\widehat{\Delta}(\mathit{exp}) = (R, T, S, P)$.

`select`, `cut`, and `gapify`: The modeling of these operations is based on a technique for symbolic XPath evaluation on summary graphs described later in this section.

`smash`: To model an instance of the `smash` operation, let n denote its template node and define $\mathit{gaps}(n) = g_1 g_2$ where g_1 and g_2 are fresh unique gap names. If $\widehat{\Delta}(\mathit{exp}) = (R, T, S, P)$ then we define $\widehat{\Delta}(\texttt{smash(}\mathit{exp}\texttt{)}) = (\{n\}, T', S, P')$, where T' and P' are copies of T and P, respectively, with the following modifications: $(n, g_1, m) \in T'$ for each $m \in R$, $(n, g_2, n) \in T'$, and $n \in \mathit{open}(P(g_i))$ and $n \in \mathit{tgaps}(P(g_i))$ for $i = 1, 2$.

`cast` and `get`: The difficult part of modeling these operations is to construct a summary graph SG_D for a given DTD D such that $\mathcal{L}(SG_D) = \mathcal{L}(D)$. We show below how this can be achieved.

All transfer functions can be shown to be monotone.

Once the summary graphs are constructed, the `analyze` invocations are checked using a variation of the validation algorithm from [16], which validates the summary graph for the XML expression relative to the DTD. The original algorithm works on non-normalized summary graphs and DSD2 schemas, but it is easily adjusted to the present setting. This is a conservative analysis of the summary graph: if it returns “valid”, then it is guaranteed that all XML templates at that point are valid at runtime; otherwise, a useful error message is provided.

To check that `plug` invocations always succeed, we inspect the associated summary graphs as in [16]. To check that XPath expressions in `select`, `cut`, and `gapify` invocations may potentially hit some nodes, we inspect the status maps that are generated by the symbolic evaluation presented later.

Using similar arguments as in [16], the theoretical worst-case complexity of the entire analysis can be shown to be $O(n^8)$ where n is the total size of the program and the relevant DTD schemas. Despite this high theoretical bound, the analysis appears efficient in practice, as shown in Section 6.

Summary Graphs for XML Template Constants

Given a template constant *xml*, we wish to construct a summary graph SG_{xml} such that $unfold(SG_{xml}) = \{xml\}$. This is trivial for the non-normalized summary graphs in [16] where each template constant corresponds to an individual summary graph node. For normalized summary graphs, the desired summary graph $SG_{xml} = (R, T, S, P)$ is the least one that satisfies the following constraints:

- For each element <$e\ldots$>$d_1 \ldots d_k$</e> in the template, let n denote the template node of the contents $d_1 \ldots d_k$ and define $gaps(n) = g_1 \ldots g_k$ where $g_i = h_i$ if $d_i =$ <[h_i]> and otherwise g_i is a fresh unique gap name. For each i, add $(n, g_i, m_i) \in T$ where m_i is the element node or chardata node of d_i.

- For the toplevel template contents corresponding to the template node r, we define $gaps(r)$ and add template edges in the same way as for element contents, and we define $R = \{r\}$.

- For every attribute a="s" corresponding to an attribute node n, add $S(n) = \{s\}$, and similarly for chardata.

- For every attribute gap a=[g] corresponding to an attribute node n, add $n \in open(P(g))$ and $agaps(P(g)) = \{\mathsf{OPEN}\}$.

- For every template gap <[g]> belonging to a template node n, add $n \in open(P(g))$, $tgaps(P(g)) = \{\mathsf{OPEN}\}$.

- Unless defined otherwise above, $agaps(P(g))$ and $tgaps(P(g))$ are set to $\{\mathsf{CLOSED}\}$.

Converting DTD Schemas to Summary Graphs

A given DTD D referred to from the program being analyzed is in Section 4 associated a subset of the summary graph nodes. In the following, we derive a summary graph $SG_D = (R, T, S, P)$ using those nodes such that $\mathcal{L}(SG_D) = \mathcal{L}(D)$, that is, it is an exact model of D.

As for template constants, we construct the summary graph as the least solution to a set of constraints. The algorithm runs in linear time in the size of D. First, define $R = \{r\}$ where r is the element node of the `DOCTYPE` root element. For all $g \in G$, define $agaps(P(g)) = tgaps(P(g)) = \{\mathsf{CLOSED}\}$.

For each `ELEMENT` corresponding to an element node p, we let $n = contents(p)$ and encode the content model recursively in its structure using the template node n associated to each sub-expression. For each rule, g is a fresh gap name, and unless otherwise mentioned, $gaps(n) = g$:

`#PCDATA`: Add $(n, g, m) \in T$ where m is the chardata node for `#PCDATA`. Let $S(m) = \Sigma^*$.

`ANY`: As the rule for `#PCDATA`, but we also add $(n, g, m) \in T$ for each element node m.

`EMPTY`: For the empty content model, we let $gaps(n) = \epsilon$.

E: A single element name E is modeled by adding $(n, g, m) \in T$ with m being the element node of E.

`(`C_1`,`...`,`C_k`)`: A sequence corresponds to defining $gaps(n) = g_1 \cdots g_k$ and $(n, g_i, m_i) \in T$ where m_i is the template node of C_i.

`(`C_1`|`...`|`C_k`)`: A choice corresponds to adding $(n, g, m) \in T$ for each template node m of $C_1, \ldots, C_k$.

`(`C`)?`: For optional contents, let $(n, g, m) \in T$ for the template node m of C and add $m \in removed(P(g))$.

`(`C`)+`: A repetition of one or more items is encoded by defining $gaps(n) = g_1 g_2$ and adding $(n, g_1, m) \in T$ with m being the template node of C, $(n, g_2, n) \in T$, and $n \in removed(P(g_2))$.

`(`C`)*`: As the previous rule but adding $n \in removed(P(g_1))$.

For each `ATTLIST` describing an attribute A corresponding to an attribute node n, let $S(n) = \{s_1, \ldots, s_k\}$ if the valid values of A are described by an enumeration $s_1, \ldots, s_k$, and let $S(n) = \Sigma^*$ otherwise. If A is declared as `#IMPLIED`, then add $n \in removed(P(g))$ for some g.

This construction indicates that our analysis can be extended to more expressive schema languages than DTD. For example, we immediately support unrestricted regular expressions as content models and arbitrary regular languages for describing valid character data and attribute values; however, we defer a full generalization to, for example, the DSD2 schema language, which, as previously mentioned, our algorithm for validating summary graphs relative to schemas already supports.

Symbolic XPath Evaluation

To model the XML operations that involve XPath, we symbolically evaluate a given XPath location path p on a summary graph $SG = (R, T, S, P)$. This evaluation is expressed by a function *eval* that maps (SG, p) into a status map of the form $N_\mathcal{E} \cup N_\mathcal{A} \cup N_\mathcal{C} \to \mathcal{S}$ where $\mathcal{S} = \{\mathsf{ALL}, \mathsf{SOME}, \mathsf{DEFINITE}, \mathsf{NONE}, \mathsf{DONTKNOW}\}$. For a concrete unfolding $x \in \mathcal{L}(SG)$, a given element, attribute, or chardata node n from SG may correspond to a number of XML tree nodes in x. A concrete evaluation of p on x may select only some of those nodes. Informally, the possible values of $eval(SG, p)(n)$ have the following meaning:

ALL: in every unfolding, every tree node corresponding to n is selected by p;

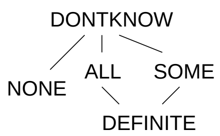
SOME: in every unfolding, at least one tree node corresponding to n is selected by p;

DEFINITE: the conditions for ALL and SOME are both satisfied;

NONE: in every unfolding, no tree node corresponding to n is selected by p; and

DONTKNOW: none of the above can be determined.

These five values form a partial order, $\sqsubseteq$, with DONTKNOW as top element and ALL and SOME above DEFINITE:



To initialize the XPath evaluation, we modify SG by introducing a dummy root element *root* and a dummy template node t where $contents(root) = t$ and $gaps(t) = g$, adding $\{(root, g, n) \mid n \in R\}$ to T, and changing R to $\{root\}$. In the following, SG refers to this modified summary graph.

We define *eval* as an evaluation of the given location path relative to an initial status map σ_0^{SG}:

$$eval(SG, p) = \big(path_p^{SG}(\sigma_0^{SG})\big)[root \mapsto \mathsf{NONE}]$$

$$\sigma_0^{SG}(n) = \begin{cases} \mathsf{DEFINITE} & \text{if } n = root \\ \mathsf{NONE} & \text{otherwise} \end{cases}$$

The notation $f[x \mapsto y]$ denotes the function that is equal to f except that it maps x to y. A location path $p = s_1/\ldots/s_k$ is evaluated compositionally on each step:

$$path_{s_1/\ldots/s_k}^{SG} = step_{s_k}^{SG} \circ \cdots \circ step_{s_1}^{SG}$$

where a single step $s = axis\texttt{::}test\texttt{[}pred\texttt{]}$ is evaluated by considering each of the three constituents:

$$step^{SG}_{axis\texttt{::}test\texttt{[}pred\texttt{]}} = filter^{SG}_{pred} \circ filter^{SG}_{test} \circ move^{SG}_{axis}$$

Recall that *axis* is either `child`, `descendant-or-self`, or `attribute`, *test* is either `text()`, `node()`, `*`, or an element or attribute name, and *pred* is either a nested location path or an expression of another type.

We define a reachability relation, $\rightsquigarrow$, as the reflexive transitive closure of the following rules: $n \rightsquigarrow contents(n)$, $n \rightsquigarrow a$ for all $a \in attr(n)$, and $n \rightsquigarrow m$ for all $(n, g, m) \in T$. The function $move^{SG}_{axis}$ is then defined as follows:

$$move^{SG}_{axis}(\sigma)(n) = \begin{cases} \mathsf{ALL} & \text{if } \forall m : root \rightsquigarrow m \ \wedge \ m \overset{?}{\triangleright}_{axis} n \Rightarrow \sigma(m) \sqsubseteq \mathsf{ALL} \\ \mathsf{SOME} & \text{if } \exists m : root \rightsquigarrow m \ \wedge \ m \overset{!}{\triangleright}_{axis} n \ \wedge \ \sigma(m) \sqsubseteq \mathsf{SOME} \\ \mathsf{DEFINITE} & \text{if the conditions for } \mathsf{ALL} \text{ and} \\ & \quad \mathsf{SOME} \text{ are both satisfied} \\ \mathsf{NONE} & \text{if } \forall m : root \rightsquigarrow m \ \wedge \ m \overset{?}{\triangleright}_{axis} n \Rightarrow \sigma(m) = \mathsf{NONE} \\ \mathsf{DONTKNOW} & \text{otherwise} \end{cases}$$

The relation $m \overset{?}{\triangleright}_{axis} n$ is satisfied if there exists an unfolding starting from m and considering only the nodes corresponding to *axis* such that n is involved. Conversely, $m \overset{!}{\triangleright}_{axis} n$ means that *every* unfolding involves n if starting from m and considering only the nodes that correspond to *axis*. We omit the formal definition, which is straightforward but tedious.

The function $filter^{SG}_{test}$ changes the status of a node n to NONE if the kind and name of n does not match *test*.

If *pred* is a location path p', then $filter^{SG}_{pred}$ will recursively apply *path* as follows: Define $\sigma'' = path^{SG}_{p'}(\sigma')$ where $\sigma'(n) = \sigma(n)$ and $\sigma'(m) = \mathsf{NONE}$ for $m \neq n$, and:

$$filter^{SG}_{p'}(\sigma)(n) = \begin{cases} \mathsf{NONE} & \text{if } \sigma(n) = \mathsf{NONE} \text{ or } \forall m : \sigma''(m) = \mathsf{NONE} \\ \sigma(n) & \text{if } \exists m : \sigma''(m) \sqsubseteq \mathsf{SOME} \\ \mathsf{DONTKNOW} & \text{otherwise} \end{cases}$$

This definition can be extended to also precisely model negated predicates and unions of node sets. If *pred* is not a location path, then $filter^{SG}_{pred}$ changes the status of a node n to DONTKNOW unless its status is already NONE.

From this definition of *eval*, we can model `select`:

$$\begin{aligned} &\widehat{\Delta}(exp\texttt{.select(}p\texttt{)}) = \\ &\quad \big(\{t\}, \\ &\quad T \cup \{(t, g, c)\} \cup \{(t, g, n) \mid n \in HITS \cap N_{\mathcal{E}}\}, \\ &\quad S\Big[c \mapsto \bigcup_{m \in HITS \cap (N_{\mathcal{C}} \cup N_{\mathcal{A}})} S(m)\Big], \\ &\quad P'[g \mapsto (\emptyset, REMOVE, \{\mathsf{CLOSED}\}, \{\mathsf{CLOSED}\})]\big) \end{aligned}$$

The nodes t and c are the associated template node and chardata node, respectively, where $gaps(t) = g$ for a fresh gap name g. The summary graph $SG = (R, T, S, P)$ is obtained from $\widehat{\Delta}(exp)$ by adding the dummy root, as explained above. The sets $HITS$ and $REMOVE$ are defined by:

$$HITS = \{n \mid eval(SG, p)(n) \neq \mathsf{NONE}\}$$

$$REMOVE = \begin{cases} \emptyset & \text{if } \forall n \in HITS : eval(SG, p)(n) \sqsubseteq \mathsf{SOME} \\ \{t\} & \text{otherwise} \end{cases}$$

Intuitively, the t node collects all nodes that may be selected, and the c node collects the values of selected attributes and character data. The gap g may be removed in t if it is possible that no element nodes are selected. The modified gap presence map P' models the disappearance of gaps in fragments that are not selected:

$$\begin{aligned} P'(h) = \big(&open(P(h)) \backslash DEAD, \\ &removed(P(h)) \backslash DEAD, \\ &GAPS_{tgaps}(h), \\ &GAPS_{agaps}(h)\big) \end{aligned}$$

$$GAPS_{\gamma}(h) = \begin{cases} \{\mathsf{OPEN}\} & \text{if } \gamma(P(h)) = \{\mathsf{OPEN}\} \ \wedge \ open(P(h)) \subseteq LIVE \\ \{\mathsf{CLOSED}\} & \text{if } \gamma(P(h)) = \{\mathsf{CLOSED}\} \ \vee \ open(P(h)) \subseteq DEAD \\ \{\mathsf{OPEN}, \mathsf{CLOSED}\} & \text{otherwise} \end{cases}$$

where, informally, $LIVE \subseteq N$ contains a node n if for every unfolding of SG all instances of n are certain to be retained by the operation; and similarly, $DEAD$ contains the nodes that are certain to be removed.

The modeling of `gapify` is defined similarly:

$$\begin{aligned} &\widehat{\Delta}(exp\texttt{.gapify}(p, g)) = \\ &\quad \big(R, \\ &\quad T \setminus \{(n, h, m) \in T \mid m \in ALL\} \\ &\qquad \cup \{(n, h, t) \mid (n, h, m) \in T \wedge m \in HITS\}, \\ &\quad S[n \mapsto \emptyset \text{ for each } n \in ALL \cap (N_{\mathcal{C}} \cup N_{\mathcal{A}})], \\ &\quad P'\big[g \mapsto (open(P(g)) \cup \{t\} \cup (HITS \cap N_{\mathcal{A}}), \\ &\qquad\qquad removed(P(g)), \\ &\qquad\qquad merge(ANY_{N_{\mathcal{E}} \cup N_{\mathcal{C}}}, tgaps(P(g))), \\ &\qquad\qquad merge(ANY_{N_{\mathcal{A}}}, agaps(P(g))))\big]\big) \end{aligned}$$

where t is the associated template node, $gaps(t) = g$, and ALL and ANY are defined by:

$$ALL = \{n \mid eval(SG, p)(n) \sqsubseteq \mathsf{ALL}\}$$

$$ANY_M = \begin{cases} \{\mathsf{OPEN}\} & \text{if } \exists n \in M : eval(SG, p)(n) \sqsubseteq \mathsf{SOME} \\ \{\mathsf{CLOSED}\} & \text{if } \forall n \in M : eval(SG, p)(n) = \mathsf{NONE} \\ \{\mathsf{OPEN}, \mathsf{CLOSED}\} & \text{otherwise} \end{cases}$$

and the function *merge* is the same as in [16]:

$$merge(\gamma_1, \gamma_2) = \begin{cases} \{\mathsf{OPEN}\} & \text{if } \gamma_1 = \{\mathsf{OPEN}\} \vee \gamma_2 = \{\mathsf{OPEN}\} \\ \gamma_1 \cup \gamma_2 & \text{otherwise} \end{cases}$$

Intuitively, the t node represents the newly constructed template gaps. Template edges into nodes that are certain to be selected are removed, and new template edges to the t node are added in place of all potentially selected nodes. The string edge map is modified by removing all strings that belong to chardata and attribute nodes that are certain to be selected. For the gap presence of g we add t and all potentially selected attribute nodes to the *open* component; for the *tgaps* component, we consider the possibility that a template gap has been added; and similarly for the *agaps* component for attribute gaps. For other gaps, we use P' as in `select`.

The `cut` operation can be modeled in the same way as `select`. However, it is possible to increase precision for both `cut` and `gapify` by also modeling the property of the semantics of these operations that an XML tree node is never considered selected if an ancestor is. We model this property by inserting an application of a function *sharpen* to the result of each application of $eval(SG, p)$. Intuitively, *sharpen* traverses SG from the roots and, for instance, converts ALL to NONE for a node n if it is able to determine that n has an ancestor of status ALL in every possible unfolding. Due to the limited space, we omit the full definition.

6 Implementation and Experiments

We have developed a prototype implementation of the runtime system and the analysis algorithms. Our experiments mainly focus on exposing the expressive power of our language design and the feasibility and precision of our analysis.

We have collected a number of small benchmark applications, inspired by typical tasks performed in other languages such as XSLT, XQuery, JDOM, and XDuce.

The `ToUpper` benchmark is from Section 3 and changes all XML recipe titles to upper case. The `AddrBook1` benchmark is the standard XDuce example, and the `AddrBook2` benchmark is a variation with a more realistic XML design. The `BankServlet` is a Servlet that produces an XHTML account summary from an XML database. The `Recipes` benchmark emulates an XSLT stylesheet producing XHTML from XML recipes; however, our version statically guarantees that the output is valid XHTML. The `Article` benchmark manipulates articles represented in XML. The `BCedit` benchmark from [35] is originally based on JDOM and implements a graphical editor on XML business cards. The `Tree` benchmark implements all queries in the corresponding XQuery use case [12]. Finally, `HTML2latex` is a benchmark from the ℂDuce project [3].

Example	Lines	Input	Output	Time	False Errors
`ToUpper`	26	25	25	3.0	0
`AddrBook1`	32	4	3	3.6	0
`AddrBook2`	17	5	4	2.8	0
`BankServlet`	65	5	1,201	5.4	0
`Recipes`	137	25	1,201	44.6	0
`Article`	123	8	1,235	5.6	0
`BCedit`	183	9	9	6.5	0
`Tree`	73	15	24	5.2	0
`HTML2latex`	159	1,201	0	11.0	0

In this table, “Lines” is the the number of lines in a desugared self-contained application, “Input” is the total number of lines of the DTD schemas involved in `cast` and `get` operations, “Output” is the the total number of lines of the DTD schemas involved in `analyze` operations, and the analysis time in measured in seconds. The memory consumption ranges from 40 to 270 MB. The precision of our analysis is reflected in the number of false errors flagged during analysis, which in all cases turns out to be zero. Furthermore, during the programming of the examples, the analysis found several actual errors that were subsequently corrected.

All experiments are performed on a 1 GHz Pentium III with 1 GB RAM running Linux and J2SE. The source for all benchmarks is available from `http://www.brics.dk/Xact/`.

The analysis is seen to be quite efficient on a wide range of benchmarks. On a subjective note, the XACT language is easy to use. It often produces programs that are as concise and readable as more specialized notations. For example, the six queries themselves in the `Tree` benchmark are written in 33 lines of code, compared to 45 lines in XQuery. At the same time our solutions are statically validated, in stark contrast to e.g. XSLT and JDOM solutions.

7 Conclusion

We have presented the XACT system, which provides a high-level approach for manipulating XML data in Java and a program analysis for statically validating the generated documents. Experiments indicate that the language design allows a concise programming style and that the analysis is efficient enough to be practically feasible.

In our future work, we will attempt to generalize the present results in various directions: We believe that XSLT stylesheets can be statically validated with the summary graph technique presented here and that it is possible to use a more powerful schema language, such as DSD2, as XML types. This will include support for XML namespaces, which is not relevant when using DTD.

We plan to integrate XACT into frameworks for making Web services, in particular JWIG and Servlets, and to make the system available as a stand-alone package for XML transformation in Java.

References

[1] Amazon.com. Amazon web services. `http://associates.amazon.com/exec/panama/associates/join/developer/resources.html`, 2002.

[2] Vidur Apparao et al. Document Object Model (DOM) level 1 specification, October 1998. W3C Recommendation. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[3] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: a white paper, October 2002. Presented at Programming Language Technologies for XML, PLAN-X.

[4] Alexandru Berlea and Helmut Seidl. Transforming XML documents using fxt. *Computing and Information Technology, Special Issue on Domain-Specific Languages*, 10(1):19–35, 2002.

[5] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, June 2001.

[6] Scott Boag et al. XQuery 1.0: An XML query language, November 2002. W3C Working Draft. `http://www.w3.org/TR/xquery/`.

[7] Scott Boag et al. Transformation API for XML. `http://xml.apache.org/xalan-j/trax.html`, 2003.

[8] Ronald Bourret. XML data binding resources, February 2003. `http://www.rpbourret.com/xml/XMLDataBinding.htm`.

[9] Claus Brabrand, Michael I. Schwartzbach, and Mads Vanggaard. The metafront system: Extensible parsing and transformation. In *Proc. 3rd ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications, LDTA '03*, April 2003.

[10] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (second edition), October 2000. W3C Recommendation. `http://www.w3.org/TR/REC-xml`.

[11] David Brownell. *SAX2*. O'Reilly & Associates, January 2002.

[12] Don Chamberlin et al. XML Query use cases, November 2002. W3C Working Draft. `http://www.w3.org/TR/xmlquery-use-cases/`.

[13] Aske Simon Christensen and Anders Møller. *JWIG User Manual*. BRICS, Department of Computer Science, University of Aarhus, June 2002. Notes Series NS-02-6. Available from `http://www.brics.dk/JWIG/manual/`.

[14] 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.

[15] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Efficient representation of XML templates, 2003. In preparation.

[16] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 2003. To appear.

[17] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. International Static Analysis Symposium, SAS '03*, June 2003.

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

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

[20] Richard Connor, David Lievens, Fabio Simeoni, Steve Neely, and George Russell. Projector – a partially typed language for querying XML, October 2002. Presented at Programming Language Technologies for XML, PLAN-X.

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

[22] Exolab Group. Castor, 2002. `http://castor.exolab.org/`.

[23] Michael Fitzgerald. Relaxer tutorial. `http://www.relaxer.org/doc/tutorial/tutorial.html`, 2003.

[24] Vladimir Gapayev and Benjamin C. Pierce. Regular object types. In *Proc. 10th International Workshops on Foundations of Object-Oriented Languages, FOOL '03*, January 2003.

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

[26] Jason Hunter and Brett McLaughlin. JDOM, 2001. `http://jdom.org/`.

[27] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977. Springer-Verlag.

[28] Martin Kempa and Volker Linnemann. Type checking in XOBE. In *Proc. Datenbanksysteme für Business, Technologie und Web, BTW '03*, volume 26 of *LNI*, February 2003.

[29] Oleg Kiselyov and Shriram Krishnamurthi. SXSLT: Manipulation language for XML. In *Proc. 5th International Symposium on Practical Aspects of Declarative Languages, PADL '03*, January 2003.

[30] 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.

[31] Erik Meijer and Wolfram Schulte. Unifying tables, objects and documents. In submission, 2003.

[32] Erik Meijer and Mark Shields. XMλ: A functional language for constructing and manipulating XML documents. Draft. Available from `http://www.cse.ogi.edu/~mbs/pub/xmlambda/`, 1999.

[33] 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.

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

[35] Anders Møller and Michael I. Schwartzbach. The XML revolution - technologies for the future Web, December 2001. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-01-8. Available from `http://www.brics.dk/~amoeller/XML/`. Revision of BRICS NS-00-8.

[36] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, October 1999.

[37] 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*, May 2000.

[38] Steven Pemberton et al. XHTML 1.0: The extensible hypertext markup language, January 2000. W3C Recommendation. `http://www.w3.org/TR/xhtml1`.

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

[40] Fabio Simeoni, Paolo Manghi, David Lievens, Richard H. Connor, and Steve Neely. An approach to high-level language bindings to XML. *Information & Software Technology*, 44(4):217–228, 2002. Elsevier.

[41] Dan Suciu. The XML typechecking problem. *ACM SIGMOD Record*, 31, March 2002.

[42] Sun Microsystems. Java API for XML processing. `http://java.sun.com/xml/jaxp/`, 2001.

[43] Sun Microsystems. Java Servlet Specification, Version 2.3, 2001. Available from `http://java.sun.com/products/servlet/`.

[44] Sun Microsystems. JAXB, 2002. `http://java.sun.com/xml/jaxb/`.

[45] Peter Thiemann. WASH/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Proc. 4th International Symposium on Practical Aspects of Declarative Languages, PADL '02*, January 2002.

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

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

[48] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming, ICFP '99*, September 1999.

Recent BRICS Report Series Publications

RS-03-19 **Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach.** ***Static Analysis of XML Transformations in Java*****. May 2003. 29 pp.**

RS-03-18 **Bartek Klin and Paweł Sobociński.** ***Syntactic Formats for Free: An Abstract Approach to Process Equivalence*****. April 2003. 41 pp.**

RS-03-17 **Luca Aceto, Jens Alsted Hansen, Anna Ingólfsdóttir, Jacob Johnsen, and John Knudsen.** ***The Complexity of Checking Consistency of Pedigree Information and Related Problems*****. March 2003. 31 pp. This paper supersedes BRICS Report RS-02-42.**

RS-03-16 **Ivan B. Damgård and Mads J. Jurik.** ***A Length-Flexible Threshold Cryptosystem with Applications*****. March 2003. 19 pp.**

RS-03-15 **Anna Ingólfsdóttir.** ***A Semantic Theory for Value–Passing Processes Based on the Late Approach*****. March 2003. 48 pp.**

RS-03-14 **Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard.** ***From Interpreter to Compiler and Virtual Machine: A Functional Derivation*****. March 2003. 36 pp.**

RS-03-13 **Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard.** ***A Functional Correspondence between Evaluators and Abstract Machines*****. March 2003. 28 pp.**

RS-03-12 **Mircea-Dan Hernest and Ulrich Kohlenbach.** ***A Complexity Analysis of Functional Interpretations*****. February 2003. 70 pp.**

RS-03-11 **Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde.** ***Fast Partial Evaluation of Pattern Matching in Strings*****. February 2003. 14 pp. To appear in Leuschel, editor,** ***ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*****, PEPM '03 Proceedings, 2003.**

RS-03-10 **Federico Crazzolara and Giuseppe Milicia.** ***Wireless Authentication in χ-Spaces*****. February 2003. 20 pp.**

RS-03-9 **Ivan B. Damgård and Gudmund Skovbjerg Frandsen.** ***An Extended Quadratic Frobenius Primality Test with Average and Worst Case Error Estimates*****. February 2003. 53 pp.**