XML Graphs in Program Analysis

Anders Møller and Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark
{amoeller,mis}@brics.dk

Abstract

XML graphs have shown to be a simple and effective formalism for representing sets of XML documents in program analysis. It has evolved through a six year period with variants tailored for a range of applications. We present a unified definition, outline the key properties including validation of XML graphs against different XML schema languages, and provide a software package that enables others to make use of these ideas. We also survey four very different applications: XML in Java, Java Servlets and JSP, transformations between XML and non-XML data, and XSLT.

1 Introduction

Many interesting programming formalisms deal explicitly with XML documents. Examples range from domain-specific languages, such as XSLT and XQuery, to general-purpose languages, such as Java in which XML documents may be handled by special frameworks or simply as text.

When such programs are the subject of static analyses, it is necessary to obtain a formal model of sets of XML documents or fragments, typically to represent conservative approximations of the possible results at specific program points. Several such models have been proposed, mainly based on the observation that formal tree languages capture many desired properties since XML documents are essentially trees [38, 19]. For practical use, a good and versatile model aimed at static analysis must satisfy some particular requirements:

- it must capture all features in XML that are relevant for validation, not just an idealized subset – in particular, we cannot ignore attributes, character data, or interleaved contents;
- it must provide a finite-height lattice structure for use in dataflow analysis with fixed-point iteration;
- it must be able to express sets of XML documents described by common schema formalisms, such as DTD [7] and XML Schema [39];
- it must allow static validation against common schema formalisms and also navigation with XPath expressions [14]; and
- it must be fully implemented.

In this paper we describe the *XML graph* model, which has matured through substantial practical experience in building static analyses of languages that manipulate XML. We also survey four different applications, showing the versatility of the model. XML graphs are fully implemented and available in an open source software package.

2 XML Graphs

W3C's DOM [1] and XDM [17] both provide a view of XML documents as unranked, labeled, finite trees. For example, elements correspond to labeled nodes with children representing the element contents. XML graphs generalize the notion of XML trees in a number of directions:

- Character data text, attributes values, and element/attribute names are described by *regular string languages* rather than by single strings.
- In XML trees, the content of an element is always described as an ordered sequence. XML graphs add special *choice* and *interleave* nodes describing more general content models.
- Loops are permitted in XML graphs.
- Some applications (see Section 3.1) involve a notion of *gaps*, which are represented in XML graphs by a variant of choice nodes as explained below.

A single XML graph generally represents a *set* of XML documents, namely those that can be obtained by "unfolding" the XML graph, starting from a root node, and then following all possible combinations of choices and interleavings described by the choice and interleave nodes and all possible character data text, attributes values, and element/attribute names described by the regular string languages. This is all defined formally in the next section.

EXAMPLE The XML graph shown in Figure 1 represents the set of XML documents consisting of one `ul` element that contains a sequence of zero or more `li` elements, each containing a numeral (described by the regular language $[0-9]^+$): We here use a choice node and a sequence node arranged in a loop for expressing the unbounded number of `li` elements.

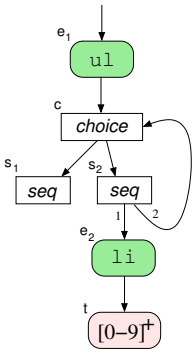

Figure 1. Example XML graph.

2.1 Definition

The various applications of XML graphs (or *summary graphs*, as they were called in earlier papers) have involved different variants, tailored for the different needs. We here present a coherent definition of XML graphs that fits closely with our newest and complete implementation.

An *XML graph*, χ, is a quintuple:

$$\chi = (\mathcal{N}, \mathcal{R}, \mathit{contents}, \mathit{strings}, \mathit{gaps})$$

The finite set $\mathcal{N} = \mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{A} \cup \mathcal{N}_\mathcal{T} \cup \mathcal{N}_\mathcal{S} \cup \mathcal{N}_\mathcal{C} \cup \mathcal{N}_\mathcal{I} \cup \mathcal{N}_\mathcal{G}$ consists of *nodes* of various kinds: *element* nodes ($\mathcal{N}_\mathcal{E}$), *attribute* nodes ($\mathcal{N}_\mathcal{A}$), *text* nodes ($\mathcal{N}_\mathcal{T}$), *sequence* nodes ($\mathcal{N}_\mathcal{S}$), *choice* nodes ($\mathcal{N}_\mathcal{C}$), *interleave* nodes ($\mathcal{N}_\mathcal{I}$), and *gap* nodes ($\mathcal{N}_\mathcal{G}$). The graph has a set of *root nodes* $\mathcal{R} \subseteq \mathcal{N}$.

The map *contents* describes the outgoing edges for the different kinds of nodes:

$$\begin{aligned} \mathit{contents} &: \mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{A} \to \mathcal{N} \\ \mathit{contents} &: \mathcal{N}_\mathcal{S} \cup \mathcal{N}_\mathcal{I} \to \mathcal{N}^* \\ \mathit{contents} &: \mathcal{N}_\mathcal{C} \cup \mathcal{N}_\mathcal{G} \to 2^\mathcal{N} \end{aligned}$$

The map $\mathit{strings} : \mathcal{N}_\mathcal{T} \cup \mathcal{N}_\mathcal{A} \cup \mathcal{N}_\mathcal{E} \to \mathcal{S}$, where $\mathcal{S}$ is a family of regular string languages over the Unicode alphabet, assigns sets of strings to nodes of certain kinds for describing text (character data or attribute values) and names of elements and attributes.

The map *gaps* describes information about gaps:

$$\mathit{gaps} : \mathcal{G} \to 2^{\mathcal{N}_\mathcal{G}} \times 2^{\mathcal{N}_\mathcal{G}} \times \Gamma \times \Gamma \times \mathcal{T}$$

where $\mathcal{G}$ is a fixed set of *gap names*, $\mathcal{T}$ is a set of *schema type names* (equipped with top and bottom), and $\Gamma = 2^{\{\mathsf{OPEN},\mathsf{CLOSED}\}}$. Let *open*, *removed*, *egaps*, *agaps*, and *type* be defined by

$$\mathit{gaps}(g) = (\mathit{open}(g), \mathit{removed}(g), \mathit{egaps}(g), \mathit{agaps}(g), \mathit{type}(g))$$

Informally, *open* and *removed* specify which nodes may contain open or removed gaps; *egaps* and *agaps* describe the presence of gaps in element contents and attributes, respectively; and *type* records the types associated with typed gaps. The value $\{\mathsf{OPEN}\}$ means that one or more gaps of the given name are present, $\{\mathsf{CLOSED}\}$ means that none are present, and $\{\mathsf{OPEN},\mathsf{CLOSED}\}$ means that the gaps are present for some unfoldings but absent for others.

Not all applications involve gaps; for those that do not, the *gaps* component is simply ignored.

To simplify validation (see Section 2.3) we require that interleave nodes never appear nested within content model descriptions nor in attribute value descriptions. (This requirement is expressed formally in [23]).

We call two XML graphs *compatible* if they agree on the values of $\mathcal{N}$, $\mathcal{G}$, $\mathcal{S}$, $\mathcal{T}$, and $\mathit{contents}(n)$ for $n \in \mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{A} \cup \mathcal{N}_\mathcal{S} \cup \mathcal{N}_\mathcal{I}$. Each family of compatible XML graphs forms a finite-height lattice using a pointwise subset ordering. This is crucial when using XML graphs in dataflow analysis.

The *language*, $\mathcal{L}(\chi)$, of an XML graph χ is a set of finite strings defined by

$$\mathcal{L}(\chi) = \{x \mid \exists n \in \mathcal{R} : \; n \Rightarrow x \; ; \; t \; ; \; a\}$$

where the *unfolding relation*, $\Rightarrow$, is defined inductively according to Figure 2. Intuitively, the relation $n \Rightarrow x \; ; \; t \; ; \; a$ holds when unfolding from node n in the XML graph may produce XML content x, text t, and attributes a. The operator $\|$ produces all possible interleavings (i.e. the shuffle) of the given XML contents; the operator $\oplus$ merges sets of attributes in all possible ways where, if two attributes have the same name then one of them overrides the other. Note that not all constituents of *gaps* are used in the definition of the unfolding relation: in XACT (Section 3.1), *egaps*, *agaps*, and *type* are used in the dataflow transfer functions.

$$\frac{n \in N_\mathcal{E} \quad s \in \mathit{strings}(n) \quad \mathit{contents}(n) \Rightarrow x \; ; \; t \; ; \; a}{n \Rightarrow \texttt{<}s\; a\texttt{>}\; x\; \texttt{</}s\texttt{>} \; ; \; \varepsilon \; ; \; \emptyset} \; [\text{element}]$$

$$\frac{n \in N_\mathcal{T} \quad s \in \mathit{strings}(n)}{n \Rightarrow s \; ; \; s \; ; \; \emptyset} \; [\text{text}]$$

$$\frac{n \in N_\mathcal{A} \quad s \in \mathit{strings}(n) \quad \mathit{contents}(n) \Rightarrow x \; ; \; t \; ; \; a \quad t \neq \emptyset}{n \Rightarrow \varepsilon \; ; \; \varepsilon \; ; \; s\texttt{="}t\texttt{"}} \; [\text{attribute}]$$

$$\frac{n \in N_\mathcal{C} \cup N_\mathcal{G} \quad m \in \mathit{contents}(n) \quad m \Rightarrow x \; ; \; t \; ; \; a}{n \Rightarrow x \; ; \; t \; ; \; a} \; [\text{choice / gap}]$$

$$\frac{\begin{array}{c} n \in N_\mathcal{S} \quad \mathit{contents}(n) = m_1 \cdots m_k \\ m_i \Rightarrow x_i \; ; \; t_i \; ; \; a_i \qquad a \in a_1 \oplus \cdots \oplus a_k \end{array}}{n \Rightarrow x_1 \cdots x_k \; ; \; t_1 \cdots t_k \; ; \; a} \; [\text{sequence}]$$

$$\frac{\begin{array}{c} n \in N_\mathcal{I} \quad \mathit{contents}(n) = m_1 \cdots m_k \quad m_i \Rightarrow x_i \; ; \; t_i \; ; \; a_i \\ x \in x_1 \parallel \cdots \parallel x_k \qquad a \in a_1 \oplus \cdots \oplus a_k \end{array}}{n \Rightarrow x \; ; \; \varepsilon \; ; \; a} \; [\text{interleave}]$$

$$\frac{n \in N_\mathcal{G} \quad n \in \mathit{open}(g)}{n \Rightarrow \texttt{<[}g\texttt{]>} \; ; \; \emptyset \; ; \; \emptyset} \; [\text{open content gap}]$$

$$\frac{n \in N_\mathcal{A} \quad s \in \mathit{strings}(n) \quad \mathit{contents}(n) \in \mathit{open}(g)}{n \Rightarrow \varepsilon \; ; \; \varepsilon \; ; \; s\texttt{=[}g\texttt{]}} \; [\text{open attribute gap}]$$

$$\frac{n \in N_\mathcal{G} \quad n \in \mathit{removed}(g)}{n \Rightarrow \varepsilon \; ; \; \emptyset \; ; \; \emptyset} \; [\text{removed content gap}]$$

$$\frac{n \in N_\mathcal{A} \quad \mathit{contents}(n) \in \mathit{removed}(g)}{n \Rightarrow \varepsilon \; ; \; \varepsilon \; ; \; \emptyset} \; [\text{removed attribute}]$$

Figure 2. Inference rules for unfolding of XML graphs.

EXAMPLE The XML graph from Figure 1 can be described formally as follows.

$$\mathcal{N} = \{\mathsf{e}_1, \mathsf{e}_2, \mathsf{s}_1, \mathsf{s}_2, \mathsf{c}, \mathsf{t}\}$$
$$\mathcal{R} = \{\mathsf{e}_1\}$$
$$\mathit{contents} = [\mathsf{e}_1 \mapsto \mathsf{c},\ \mathsf{e}_2 \mapsto \mathsf{t},\ \mathsf{s}_1 \mapsto \varepsilon,\ \mathsf{s}_2 \mapsto \mathsf{e}_2\,\mathsf{c},\ \mathsf{c} \mapsto \{\mathsf{s}_1, \mathsf{s}_2\}]$$
$$\mathit{strings} = [\mathsf{e}_1 \mapsto \{\texttt{ul}\},\ \mathsf{e}_2 \mapsto \{\texttt{li}\},\ \mathsf{t} \mapsto \mathcal{L}([0-9]^+)]$$
$$\mathit{gaps} = []$$

Its language contains, for example, these three XML documents:

```
<ul></ul>
<ul><li>42</li></ul>
<ul><li>42</li><li>87</li></ul>
```

We implicitly assume that entity references have been expanded, treat CDATA sections as plain character data, and ignore attribute order, processing instructions, and comments, since these features are irrelevant for validation. XML namespaces are handled by expanding qualified names to the form {*URI*}*localname*.

2.2 Relations to Other Formalisms

Clearly, the notion of XML graphs is closely connected to, in particular, RELAX NG [15], regular expression types [19] (as explained in [8] for an early variant of XML graphs), and regular tree grammars [38]. Intuitively, an XML graph is essentially a graphical representation of those formalisms. However, loops in XML graphs may lead to content models that are not necessarily regular but generally context-free.

The main reason for using XML graphs instead of these alternatives is that XML graphs naturally form a finite-height lattice, as explained above. Additionally, to be able to express schemas written in XML Schema, we cannot ignore text, attributes, or interleaved content models. Finally, maintaining the gap information is important in, for example, the XACT program analyzer (see Section 3.1).

In [23], a language called Restricted RELAX NG is defined as a subset of RELAX NG. The subset limits expressiveness to single-type tree grammars [38], prohibits context sensitive attribute patterns, and limits the use of interleave patterns to top-level content models. Every schema written in XML Schema can be converted into a Restricted RELAX NG schema that validates the same set of documents, and Restricted RELAX NG is much simpler than the full XML Schema language. Moreover, we can validate XML graphs against Restricted RELAX NG schemas more easily than if using the full RELAX NG language.

2.3 Operations on XML Graphs

The various applications of XML graphs involve a number of interesting operations.

Representation of XML documents, templates, and schemas XML documents are merely special cases of XML graphs. XML templates, which are used in XACT (Section 3.1), add various kinds of gaps, which can be represented as gap nodes in XML graphs. Additionally, every schema written in DTD, XML Schema, or Restricted RELAX NG can be converted into an equivalent XML graph[1].

[1] A few obscure features in XML Schema datatypes go beyond regular languages; if the need should ever arise, these can be accommodated by augmenting the *strings* map. For technical details, see [23, 26].

Closure properties A central operation when XML graphs are used in dataflow analysis is computing the least upper bound of two compatible XML graphs, which is trivial by the definition in Section 2.1.

Note that XML graphs are also closed under, in particular, language union: simply rename nodes to avoid conflicts and then join the root sets. However, we have never encountered a need for performing this operation in practice.

Validation All our applications of XML graphs involve validation, that is, checking whether or not every XML document in $\mathcal{L}(\chi)$ for a given XML graph χ is valid relative to a given schema. Our algorithm, which is explained in [23], heavily exploits the restrictions in Restricted RELAX NG. It works much like an ordinary XML Schema processor by recursively traversing χ, starting at the roots, and for each element, attribute, or text node checking the constraints specified by the schema. This is done by encoding content models and schema definitions as finite string automata over a common vocabulary and checking inclusion of their languages. Loops are handled coinductively using memoization. Compared to validation algorithms based on more powerful models, such as [19] or [22], this approach exploits the single-type tree language property of the schema to obtain a simpler algorithm and to provide more informative error messages.

XPath evaluation XPath is often used for navigating in XML trees. The ordinary semantics of XPath expressions can be generalized from working on XML trees to XML graphs. Given an XPath location path p and an XML graph χ, we have an algorithm that can approximate, for each node $n \in \mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{A} \cup \mathcal{N}_\mathcal{T}$ in χ, whether or not the corresponding element, attribute, or text is definitely or maybe selected by p in $\mathcal{L}(\chi)$. This is particularly useful when analyzing XACT programs. However, it can also be used for extending the validator to check element prohibitions (e.g., that `form` elements cannot be nested in XHTML).

Additionally, the XACT analyzer (Section 3.1) models a range of basic operations on XML templates as transfer functions on XML graphs. An example is the `plug` operation, which is used for inserting XML templates or strings into gaps in other XML templates.

2.4 Implementation

The Java library `dk.brics.schematools` [26] consisting of 16,000 lines of code provides the following functionality:

- Representation of XML graphs, including various convenience methods for building, traversing, and storing XML graphs.
- Representation of schemas written in Restricted RELAX NG, including conversion from DTD and XML Schema and to XML graphs.
- Validation of XML graphs relative to Restricted RELAX NG schemas, with useful messages when invalidity is detected.
- Evaluation of XPath location path expressions on XML graphs.
- A command-line interface for performing validation and conversion for the different formalisms, as a supplement to the API.

Independently of XML graphs, the schema conversion ability fills a niche: Sun's RELAX NG Converter [21] supports conversion from XML Schema to RELAX NG but has several deficiencies; Trang [11] supports approximating conversion in the other direction only (in addition to supporting DTD). By combining schema conversion with validation, `dk.brics.schematools` can check language inclusion between schemas written in XML Schema.

3 Four Applications of XML Graphs

XML graphs have proved to be a useful tool in several applications of which we survey four examples selected from the full range [5, 9, 25, 23, 6, 36, 3, 24, 35]. They all use different aspects of the package and the languages they consider span a wide spectrum. Despite their differences, each application follows a common pattern. First, a flow analysis is performed, which of course depends closely on the particular source language. Second, XML graphs are constructed for the interesting program points; again, the techniques for doing this depends on the application domain. Third, the resulting XML graphs are analyzed, generally using the static validation tool.

3.1 XACT

The XACT language extends Java with domain-specific support for manipulating XML documents [25, 23]. It is available in an open source implementation from `http://www.brics.dk/Xact/`.

It is based of the notion of XML *templates*, which contain named *gaps*. Templates may by plugged together and may be deconstructed in various manners guided by XPath expressions. The templates are implemented as an immutable datatype in a Java framework. A preprocessor adds a layer of domain-specific syntax. A comparison between XACT and other languages for XML manipulation is presented in [37].

The following is an example of an XACT program that generates an XHTML presentation of a phone list extracted from an XML collection of business cards, in a way that exhibits the various language features:

```
import dk.brics.xact.*;

public class PhoneList {
  @DefaultXPathNamespace
  public static final String b =
      "http://businesscard.org";

  @DefaultConstantNamespace
  public static final String h =
      "http://www.w3.org/1999/xhtml";

  @Namespace
  public static final String s =
      "http://www.w3.org/2001/XMLSchema";

  public @Type("h:html[s:string TITLE, h:Flow MAIN]") XML
      wrapper;

  public @Type("h:html") XML
      transform(@Type("b:cardlist") XML cardlist) {
    return wrapper.plug("TITLE", "My Phone List")
                  .plug("MAIN", makeList(cardlist));
  }

  private XML makeList(XML x) {
    XML r =  [[<ul><[CARDS]></ul>]];
    for (XML c : x.select("card[phone]"))
      r = r.plug("CARDS",
                 [[<li>
                     <b><{ c.select("name/text()") }></b>,
                     phone: <{c.select("phone/text()") }>
                   </li>
                   <[CARDS]>]]);
    return r.close();
  }

  private void setDefaultWrapper(String color) {
    wrapper = [[<html>
                  <head>
                    <title><[s:string TITLE]></title>
                  </head>
                  <body bgcolor=[s:string COLOR] >
                    <h1><[s:string TITLE]></h1>
                    <[h:Flow MAIN]>
                  </body></html>]].plug("COLOR", color);
  }

  public static void main(String[] args)
      throws java.io.IOException {
    PhoneList pp = new PhoneList();
    pp.setDefaultWrapper("white");
    XML cardlist = XML.get("cards.xml", "b:cardlist");
    XML xhtml = pp.transform(cardlist);
    System.out.println(xhtml);
  }
}
```

The static analysis challenge for XACT is to decide if the possible values of XML expressions are guaranteed to be valid according to the given (optional) XML Schema type annotations. In the above example, this includes a guarantee that the output will always be valid XHTML if the input is a valid collection of business cards.

XML Graph Construction

Since XACT is an extension of full Java, the analysis must first construct an ordinary flow graph for the Java program. This is done using the Soot framework [41]. Subsequently, we perform a standard dataflow analysis [20] but with the highly specialized lattice structure of XML graphs described in Section 2.1. The transfer functions conservatively model the abstract semantics of the template operations. While these are certainly intricate in their details, they are actually conceptually simple. The *gaps* maps of XML graphs are here used to keep track of whether gaps in the combined templates are necessarily or possibly left open by plug operations.

To handle input and cast operations, we need to model XML Schema types directly as XML graphs, using the embedding described in Section 2.3. The precision of our analysis is boosted by the fact that such types can be modeled exactly without resorting to conservative approximations.

XML Graph Analysis

After the dataflow analysis, each XML expression has associated an XML graph that describes a superset of the possible XML values that may be the results of runtime evaluation. The XACT tool may use this information to check a number of properties. Validity of annotations reduces to the static validity check described in Section 2.3. Also, for plug operations it is checked that an open gap with the given name is present in the XML template. Finally, a warning is issued if an XPath expression will always result in an empty node sequence.

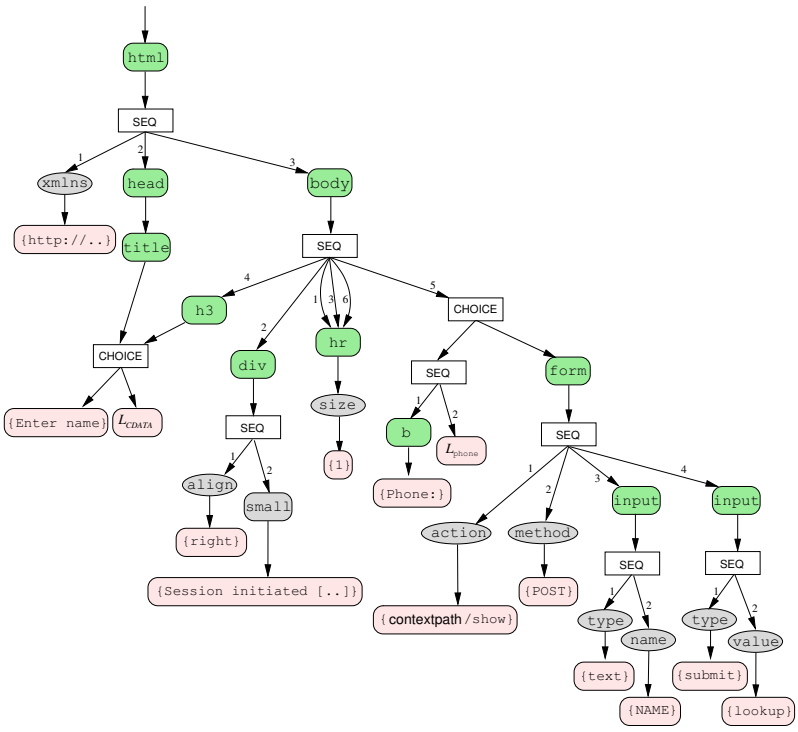

Figure 3. XML graph for servlet example.

3.2 Java Servlets and JSP

The XACT project introduces a novel extension of Java for manipulation of XML templates. In contrast, the common frameworks of Java Servlets and JSP work at a lower level where XML documents are produced one character at a time on an output stream. (JSP templates are merely converted into servlets.) This poses a substantially harder problem for static validation since now also well-formedness of the generated XML documents must be determined by the analysis. Also, the control flow of the application is more implicit since individual servlets may transfer control based on string valued URLs. The following example program shows some of the many challenges that may arise:

```
public class Entry extends javax.servlet.http.HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession();
    String url =
      response.encodeURL(request.getContextPath()+"/show");
    session.setAttribute("timestamp", new Date());
    response.setContentType("application/xhtml+xml");
    PrintWriter out = response.getWriter();
    Wrapper.printHeader(out, "Enter name", session);
    out.print("<form action=\""+url+"\" method=\"POST\">"+
              "<input type=\"text\" name=\"NAME\"/>"+
              "<input type=\"submit\" value=\"lookup\"/>"+
              "</form>");
    Wrapper.printFooter(out);
  }
}

public class Show extends javax.servlet.http.HttpServlet {
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
      throws ServletException, IOException {
    Directory directory =
      new Directory("ldap://ldap.widgets.org");
    String name = misc.encodeXML(request.getParameter("NAME"));
    response.setContentType("application/xhtml+xml");
    PrintWriter out = response.getWriter();
    Wrapper.printHeader(out, name, request.getSession());
    out.print("<b>Phone:</b> "+directory.phone(name));
    Wrapper.printFooter(out);
  }
}

public class Wrapper {
  static void printHeader(PrintWriter pw, String title,
                          HttpSession session) {
    pw.print("<html xmlns=\"http://www.w3.org/1999/xhtml\">"+
             "<head><title>"+title+"</title></head><body>"+
             "<hr size=\"1\"/>"+
             "<div align=\"right\"><small>"+
             "Session initiated ["+
             session.getAttribute("timestamp")+"]"+
             "</small></div><hr size=\"1\"/>"+
             "<h3>"+title+"</h3>");
  }

  static void printFooter(PrintWriter pw) {
    pw.print("<hr size=\"1\"/></body></html>");
  }
}
```

An obvious question is whether the `doGet` and `doPost` methods produce valid XHTML as output? In fact, we would like to verify many other properties of the above application, but they all hinge on first understanding the generated XHTML documents. In [24], a program analysis that attacks these problems is presented, based on XML graphs. The paper [35] discusses the problem of analyzing SAX stream filters, which, to some extent, can be reduced to analyzing servlets.

XML Graph Construction

Since the servlets work on strings values, we first employ an existing string analysis that computes regular languages for the possible values of all string expressions [10]. This analysis takes into account the basic control flow of the Java programs.

Well-formedness of the generated XML data is then performed by combining the theories of balanced grammars by Knuth [27] and grammar approximations by Mohri and Nederhof [34]. Finally, the transformed grammar is rather directly expressed as an XML graph, which summarizes the results of these analyses.

The XML graph for the example program is shown in Figure 3.

XML Graph Analysis

Once we have XML graphs for the possible contents of the output streams, we can, again, apply the static validation algorithm to ensure that only valid XHTML is produced. However, a more specific analysis of these graphs can answer other interesting questions about servlet applications. By analyzing the possible values of action URLs in forms it is possible to determine the control flow between individual servlets. In the above example, this knowledge will allow us to determine that the `timestamp` attribute is available in the session state when the `Show` servlet is executed. Also, by further analyzing the form fields inside the generated XHTML documents, we can guarantee that the request parameter `NAME` is always present as well.

3.3 XSugar

The XSugar project [6] provides a framework for specifying and maintaining dual syntax for XML languages. A typical situation of this kind is the XML schema language RELAX NG [15] that has and alternative, compact, non-XML syntax [13]. Other languages with dual syntax include BibTeXML [18] and the Wiki notation [29]. As an example, consider the XML document

```
<students xmlns="http://studentsRus.org/">
  <student sid="19701234">
    <name>John Doe</name>
    <email>john_doe@notmail.org</email>
  </student>
  <student sid="19785678">
    <name>Jane Dow</name>
    <email>dow@bmail.org</email>
  </student>
</students>
```

with the following alternative syntax:

```
John Doe (john_doe@notmail.org) 19701234
Jane Dow (dow@bmail.org) 19785678
```

The XML syntax may be specified in XML Schema and the alternative syntax could be specified through an XSLT stylesheet (generating plain text). However, this approach has some inherent weaknesses: consistency must be maintained between the two syntaxes, and a separate translator from alternative to XML syntax must be programmed. XSugar allows a simultaneous specification of both syntaxes in the form of a context-free grammar with dual right-hand sides. For our example, the specification looks as follows:

```
xmlns = "http://studentsRus.org/"

Name   = [a-zA-Z]+(\ [a-zA-Z]+)*
Email  = [a-zA-Z._]+\@[a-zA-Z._]+
Id     = [0-9]{8}

NL     = \r\n|\r|\n

file : [persons p] = <students> [persons p] </>

persons : [person p] [NL] [persons more] =
          [person p] [persons more]
        : =

person : [Name name] _ "(" [Email email] ")" _ [Id id] =
         <student sid=[Id id]>
           <name> [Name name] </>
           <email> [Email email] </>
         </>
```

The XSugar tool analyzes the grammar to ensure reversibility of the translation between the two versions, which involves an approximate decision procedure for ambiguity of grammars [4]. The remaining problem, which is relevant for this paper, is to decide whether the XSugar specification agrees with an original XML schema specification of the XML language, such as this one:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://studentsRus.org/"
        xmlns:s="http://studentsRus.org/"
        elementFormDefault="qualified">

  <element name="students">
    <complexType>
      <sequence minOccurs="0" maxOccurs="unbounded">
        <element ref="s:student"/>
      </sequence>
    </complexType>
  </element>

  <element name="student">
    <complexType>
      <sequence>
        <element name="name" type="s:Name"/>
        <element name="email" type="s:Email"/>
      </sequence>
      <attribute name="sid" type="s:Id"/>
    </complexType>
  </element>

  <simpleType name="Id">
    <restriction base="string">
      <pattern value="[0-9]{8}"/>
    </restriction>
  </simpleType>

  <simpleType name="Name">
    <restriction base="string">
      <pattern value="[a-zA-Z]+( [a-zA-Z]+)*"/>
    </restriction>
  </simpleType>

  <simpleType name="Email">
    <restriction base="string">
      <pattern value="[a-zA-Z._]+@[a-zA-Z._]+"/>
    </restriction>
  </simpleType>
</schema>
```

XML Graph Construction

From an XSugar specification, it is simple to extract an XML graph that describes all XML documents that can be generated by the XML productions:

- each nonterminal becomes a choice node with a child for each of its productions;

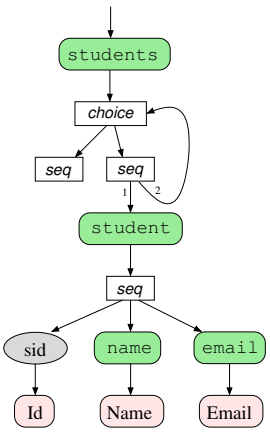

Figure 4. XML graph for XSugar example.

- a production becomes a sequence node if ordered and an interleave node if unordered, and a child node is made for each item;
- for a nonterminal item, the node is the one corresponding to the nonterminal;
- for a regular expression item, the node is a text node labeled with the regular expression. and quoted literal items and whitespace items are treated as regular expression items;
- for an element item, the node is an element node with a corresponding name and with a sequence child node describing the attributes and contents, and attributes similarly become attribute nodes.

As a simple optimization, we may omit choice nodes and sequence nodes that have exactly one child. For the student information example, the resulting XML graph is shown in Figure 4.

XML Graph Analysis

Static validation of the XSugar program is simply obtained by means of the main algorithm from Section 2.3. If we had made some mistakes, for example changed the definition of `Id` to `[0-9]{5,8}` and swapped the order of the `name` and `email` elements in the XSugar specification, the output would instead be like this:

```
*** Validation error
Source: element {http://studentsRus.org/}student at
 students.xsg line 15 column 10
Schema: students.xsd line 20 column 7
Error: invalid attribute value: sid="00000"

*** Validation error
Source: element {http://studentsRus.org/}student at
 students.xsg line 15 column 10
Schema: students.rng line 16 column 7
Error: invalid contents:
 <{http://studentsRus.org/}email/>
 <{http://studentsRus.org/}name/>
```

Clearly, such error messages are useful for locating and correcting the errors.

3.4 XSLT

An obvious challenge in the area of static validation is posed by XSLT stylesheets [12]: under the assumption that the input is valid relative to the input schema, is the output of the transformation always valid relative to the output schema? This fundamental problem was first solved and implemented in our paper [36] and generalized to XSLT 2.0 in [28]. Earlier work in this area have only provided partial solutions [2, 40, 16] or have only looked at idealized languages [33, 31, 32, 30]. As an instance of this problem, consider documents such as this:

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

which is described by this DTD schema:

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

Consider now the following XSLT stylesheet:

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

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

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

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

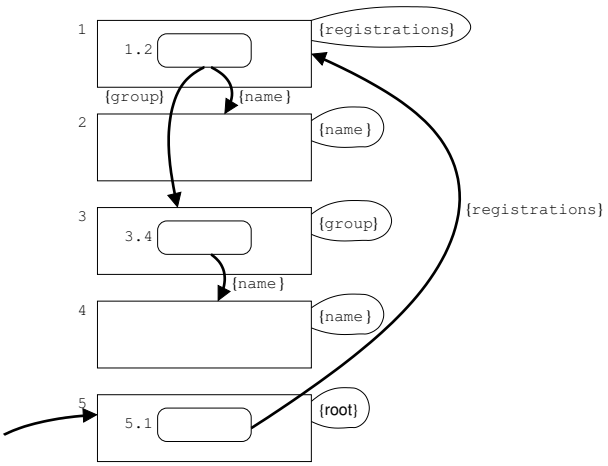

Figure 5. Flow graph for XSLT example.

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

This stylesheet transforms such documents into an XHTML presentation that may be rendered as follows:

1. John Q. Public

Widget, Inc. ®
John Doe !!!
Jane Dow
Jack Doe

2.
3. Joe Average

The question is then whether documents described by the input schema will always be transformed into valid XHTML documents?

XML Graph Construction

Before the set of possible output documents can be described, it is necessary to perform a flow analysis of the XSLT stylesheet. Specifically, we wish to determine for each `apply-templates` instruction which `template` rules may be invoked when processing some input document. In addition, we must also determine the types and names of the possible context nodes when the template is instantiated. Our algorithm defines a constraint system that defines this information, which is then computed using a fixed-point algorithm. A crucial component in this algorithm is to determine the compatibility between `select` and `match` expressions relative to the paths that are allowed by the input schema. Our algorithm is heuristic and uses conservative approximations that are guided by a extensive data mining of a collection of 603 stylesheets with a total of 187,015 lines of code written by hundreds of different authors. For our example stylesheet, the flow information is summarized as shown in Figure 5.

Based on this flow graph, the details of the stylesheet and the input schema, our algorithm constructs an XML graph that describes

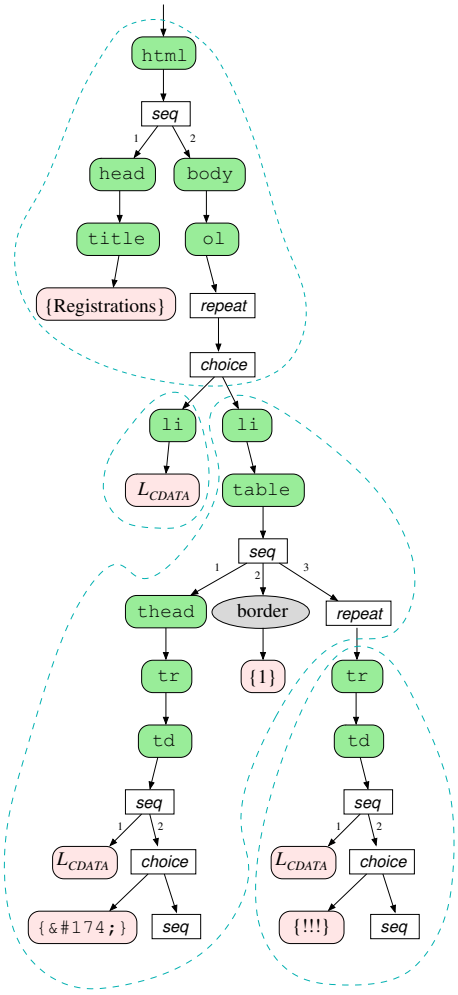

Figure 6. XML graph for XSLT example.

all possible output documents. Again, this is done using heuristics that are guided by mining the extensive stylesheet samples. The XML graph for our example is shown in Figure 6. The *repeat* abbreviate the choice–sequence loop used earlier, and the dashed lines indicate template rules in the original stylesheet.

XML Graph Analysis

Once the XML graph has been constructed, we again rely on the static validation algorithm from Section 2.3. In addition to this result, we may analyze the XML graph further to provide warnings about `select` expressions that never hit anything and template rules that are never used. These are not necessarily errors in the stylesheet, but presumably unintended by the programmer.

4 Conclusion

We have presented XML graphs as a convenient formalism for representing sets of XML documents. XML graphs have been used in a variety of analyses of programs that operate on XML data, including the languages XACT, Java Servlets and JSP, XSugar, and XSLT. The implementation is now available in an open source software package for others to use.

References

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

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

[3] Henning Böttger, Anders Møller, and Michael I. Schwartzbach. Contracts for cooperation between Web service programmers and HTML designers. *Journal of Web Engineering*, 5(1):65–89, 2006.

[4] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. Technical Report RS-06-09, BRICS, May 2006.

[5] 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.

[6] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. In *Proc. 10th International Workshop on Database Programming Languages, DBPL '05*, volume 3774 of *LNCS*, pages 27–41. Springer-Verlag, August 2005.

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

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

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

[10] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[11] James Clark. Trang. http://www.thaiopensource.com/relaxng/trang.html.

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

[13] James Clark. RELAX NG compact syntax, November 2002. OASIS. http://relaxng.org/compact.html.

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

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

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

[17] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 data model (XDM), November 2006. W3C Proposed Recommendation. http://www.w3.org/TR/xpath-datamodel/.

[18] Vidar Bronken Gundersen and Zeger W. Hendrikse. BibTeXML, 2005. http://bibtexml.sourceforge.net/.

[19] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

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

[21] Kohsuke Kawaguchi. RELAX NG converter. http://wwws.sun.com/software/xml/developers/relaxngconverter/.

[22] 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.

[23] Christian Kirkegaard and Anders Møller. Type checking with XML Schema in XACT. Technical Report RS-05-31, BRICS, 2005. Presented at Programming Language Technologies for XML, PLAN-X '06.

[24] Christian Kirkegaard and Anders Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*. Springer-Verlag, August 2006.

[25] 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.

[26] Christian Kirkegaard and Anders Møller. dk.brics.schematools, 2006. http://www.brics.dk/schematools/.

[27] Donald E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11:269–289, 1967.

[28] Søren Kuula. Practical type-safe XSLT 2.0 stylesheet authoring. Master's thesis, Department of Computer Science, University of Aarhus, 2006.

[29] Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web*. Addison-Wesley, 2001.

[30] Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. XML type checking with macro tree transducers. In *Proc. 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '05*, pages 283–294, 2005.

[31] 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.

[32] Wim Martens and Frank Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proc. 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '04*, pages 23–34, 2004.

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

[34] 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.

[35] Anders Møller. Static analysis for event-based XML processing. Technical Report RS-06-16, BRICS, October 2006.

[36] Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, 2005. Draft, accepted for ACM TOPLAS.

[37] Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. 10th International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[38] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.

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

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

[41] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot – a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference, CASCON '99*. IBM, November 1999.