XML Graphs in Program Analysis[☆]

Anders Møller and Michael Schwartzbach

{amoeller,mis}@cs.au.dk

Department of Computer Science, Aarhus University
Aabogade 34, 8200 Aarhus N, Denmark

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 the use of XML graphs for program analysis with four very different languages: Xact (XML in Java), Java Servlets (Web application programming), XSugar (transformations between XML and non-XML data), and XSLT (stylesheets for transforming XML documents).

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 [41, 20]. 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 content models;
- it must be able to express sets of XML documents described by common schema formalisms, in particular XML Schema [44];

[☆]This article is an extended version of [40] with material from [24].

- it must allow static validation against common schema formalisms and also navigation with XPath expressions [13];
- it must provide a finite-height lattice structure for use in dataflow analysis with fixed-point iteration; and
- it must be fully implemented.

In this paper we describe the *XML graph* model, which meets all these criteria. It 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.

In Section 2, XML graphs are formally defined. Section 3 describes the relation to regular expression types and the schema languages RELAX NG and XML Schema. In Section 4, we show how XML documents and schemas can be expressed as XML graphs, how to validate an XML graph relative to a schema, and how to evaluate XPath expressions on XML graphs. Our implementation is briefly described in Section 5. The survey of applications is in Section 6.

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.
- XML graphs are, as the name suggests, in general graphs, not trees. In particular, they can have cycles and multiple roots, and nodes can have indegree larger than one.
- Some applications (see Section 6.1) 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.

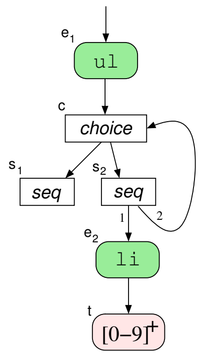


Figure 1: Example XML graph.

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 $[\texttt{0}-\texttt{9}]^+$): We here use a choice node and a sequence node arranged in a cycle for expressing the unbounded number of `li` elements.

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 $\mathcal{N} = \mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{A} \cup \mathcal{N}_\mathcal{T} \cup \mathcal{N}_\mathcal{S} \cup \mathcal{N}_\mathcal{C} \cup \mathcal{N}_\mathcal{I} \cup \mathcal{N}_\mathcal{G}$ is a disjoint union 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 notion of gaps is only relevant for some applications of XML graphs; for others, $\mathcal{N}_\mathcal{G}$ and the *gaps* component of χ can simply be ignored. Intuitively, a gap node is a named entity that represents a “missing” fragment in the XML documents. In the Xact system for manipulating XML data (see Section 6.1), this is used for modeling *XML templates* where gaps can occur in place of elements or attribute values. The main operation involving gaps in Xact is the *plug* operation, which inserts strings or XML data into the gaps, in which case we say that the gaps have been *closed*. Another operation can remove gaps; for an attribute gap this has the effect that the entire attribute is removed.

In an XML graph, the map *gaps* describes information about gaps:

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

where $\mathcal{G}$ is a finite 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

$$gaps(g) = (open(g), removed(g), egaps(g), agaps(g), type(g))$$

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.

To simplify validation (see Section 4) we require that interleave nodes never appear nested within element content model descriptions nor in attribute value descriptions. This requirement can be stated more precisely as follows. The *surface* of a node n is the set of nodes that are reachable from n, including n itself, where contents of element nodes and attribute nodes are ignored. As an example, the surface of c in the XML graph in Figure 1 is $\{\mathsf{c}, \mathsf{s}_1, \mathsf{s}_2, \mathsf{e}_2\}$. A node is a *content node* if its surface contains at least one element node or text node. In the example, which does not contain attribute nodes, every node except s_1 is a content node. We now require that (1) every node that has a child whose surface contains an interleave content node must be an element or sequence node, and (2) a sequence node whose surface contains an interleave content node must have only one content node child. This ensures that interleave nodes can only occur as roots of element content models. All XML graph constructions described in the following sections satisfy this requirement, and all operations on XML graphs preserve the property as an invariant.

Example The XML graph fragment shown in Figure 2 describes the same XML data as the following XML Schema definition:

```
<element name="E">
  <complexType>
    <all>
      <element name="F" type="..." />
      <element name="G" type="..." />
    </all>
    <attribute name="A" type="..." />
  </complexType>
</element>
```

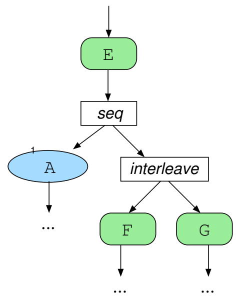


Figure 2: XML graph fragment for an XML Schema definition.

The complex type corresponds to the sequence node, and the `all` group is described by the interleave node. One can easily verify that the requirement for interleave nodes is satisfied. In the following, we formally define how XML graphs describe XML data.

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 3. 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. Because of the restrictions on interleave nodes there is no need for a similar operation on the text component, so the [interleave] rule simply yields the empty text ϵ. Note that not all constituents of *gaps* are used in the definition of the unfolding relation: in Xact (Section 6.1), *egaps*, *agaps*, and *type* are used in the dataflow transfer functions.

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 $\{URI\}localname$.

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

$$\mathcal{N} = \{\mathsf{e}_1, \mathsf{e}_2, \mathsf{s}_1, \mathsf{s}_2, \mathsf{c}, \mathsf{t}\}$$
$$\mathcal{R} = \{\mathsf{e}_1\}$$

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

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

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

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

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

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

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

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

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

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

Figure 3: Inference rules for unfolding of XML graphs.

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

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 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 $n \in \mathcal{N}_{\mathcal{E}} \cup \mathcal{N}_{\mathcal{A}} \cup \mathcal{N}_{\mathcal{S}} \cup \mathcal{N}_{\mathcal{I}}$. In other words, two

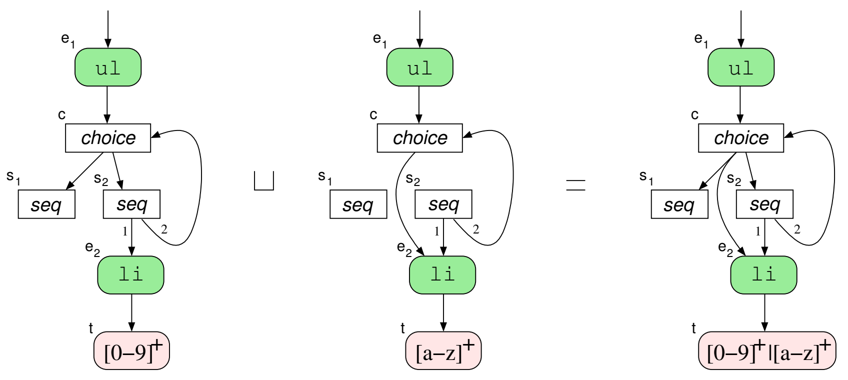


Figure 4: Merging compatible XML graphs.

compatible graphs can differ only on $\mathcal{R}$, *contents*(n) for $n \in \mathcal{N}_\mathcal{C} \cup \mathcal{N}_\mathcal{G}$ (which we call the *variable* edges), *strings*, and *gaps*. The ordering, denoted $\sqsubseteq$, on compatible XML graphs is defined pointwise on the components. In this way, each family of compatible XML graphs is a *lattice* structure, where $\sqcup$ and $\sqcap$ are respectively pointwise union and intersection [42]. These operations preserve the interleave property since this structure is fixed for compatible graphs. If we restrict *strings* to a finite codomain, then the lattice has finite height. This is crucial when using XML graphs in dataflow analysis.

Example Figure 4 shows the result of merging two compatible XML graphs using the least upper bound, denoted $\sqcup$, induced by the lattice ordering. In this case, the operation coincides with language union. Generally, least upper bound is an upper approximation of language union:

$$\mathcal{L}(\chi_1) \cup \mathcal{L}(\chi_2) \sqsubseteq \mathcal{L}(\chi_1 \sqcup \chi_2)$$

XML graphs are also closed under language union: simply rename nodes to avoid conflicts and then join the root sets (gap names should not be renamed since they may occur in the unfolding of the XML graph). However, XML graphs are not closed under language intersection since they can describe context-free content sequences (as explained in Section 3.2). Note that the full collection of XML graphs ordered by language inclusion does *not* form a finite-height lattice.

Example To demonstrate the use of gap nodes, the following Xact template can be described by the XML graph shown in Figure 5:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title><[TITLE]></title></head>
  <body bgcolor=[COLOR]>
    <h1><[TITLE]></h1>
```

```
      <[BODY]>
    </body>
  </html>
```

Here, `H` abbreviates `http://www.w3.org/1999/xhtml`. The *gaps* information is: $\textit{gaps}(\texttt{TITLE}) = (\{g_1, g_3\}, \emptyset, \{\texttt{OPEN}\}, \emptyset, \top)$. Notice that in this XML graph, some apparently superfluous choice nodes have been inserted. The XACT program analysis uses these when modeling the effect of, for example, operations that remove the contents of an element, while maintaining XML graph compatibility. We return to XACT in Section 6.1 where we show examples of how operations on templates are modeled on XML graphs.

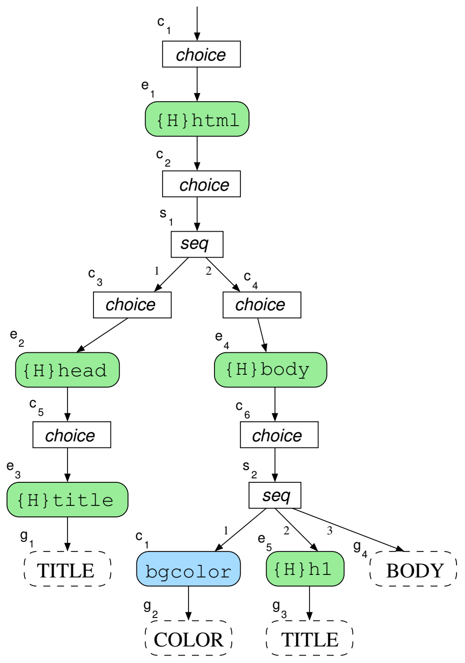


Figure 5: XML graph for an XACT template.

XML graphs are not *minimal* or *unique* representations, as in [34]: for a given XML graph, several (root) nodes may define identical sets of XML trees. For our applications, equality testing is not a central operation; instead, our representation is tuned to enable the definition of the finite-height lattices of compatible XML graphs on which least upper bounds may efficiently be computed.

We later explain the precise meaning of the fundamental operations on XML graphs: the plug operation in Section 4.1, validation against a schema in Section 4.2, and XPath evaluation in Section 4.3.

```
<card>
  <name>John Doe</name>
  <email>doe@widget.inc</email>
  <logo uri="widget.gif"/>
</card>
```

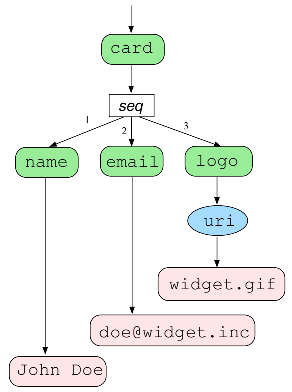


Figure 6: An XML document represented as an XML graph.

3. Relations to Other Formalisms

Clearly, the notion of XML graphs is closely connected to, in particular, RELAX NG [14], regular expression types [20] and regular tree grammars [41]. Intuitively, an XML graph is essentially a graphical representation of those formalisms.

The main reason for using XML graphs instead of these alternatives is that (compatible) XML graphs naturally form a lattice structure, 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 6.1). In the following, we provide further details about the technical differences and similarities with some of these other formalisms.

3.1. XML Documents

Every XML document can be represented as an XML graph with a singleton language, as indicated by Figure 6. As mentioned in Section 2, we are in line with other formalisms and ignore processing instructions, comments, CDATA sections, and DOCTYPE declarations as they are rarely relevant for static analysis of dynamically generated XML data. With this simplification, an XML document can be translated into an XML graph in a simple top-down manner: each XML character data node becomes an XML graph text node with a regular string language containing just the singleton string for the character contents, and each XML element node becomes an XML graph element node with a sequence node that leads to the translations of its attribute, element, or character data node contents. Thus, choice/gap/interleave nodes are never required for representing individual XML documents.

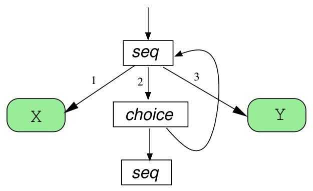


Figure 7: Non-regular contents.

3.2. Regular Expression Types

XML graphs turn out to be closely connected to the regular expression types of XDuce [19, 20]: A generalized version of regular expression types has the same expressive power as a restricted version of XML graphs [7]. Specifically, XML graphs allow for regular language restrictions on character data appearing in element contents, which is also not supported by XDuce. Also, interleave nodes do not have a counterpart in XDuce. Finally, XDuce imposes a right-linearity restriction on types, unlike XML graphs. In fact, the current implementation of XDuce forbids top-level type recursion in general.

Using XML graphs, we could construct the XML graph in Figure 7 to represent a set of XML trees with context-free but non-regular contents. Such XML graphs can occur in the analysis of Xact programs described in Section 6.1). With a linearity restriction, it would be necessary to use a conservative approximation of the possible content, such as $\mathtt{X}^*\mathtt{Y}^*$, but that could lead to spurious validation errors when the XML graph is validated against a schema.

Note that, because of the close relation between regular expression types and regular tree grammars, this discussion also applies to regular tree grammars [15, 41].

Regular expression types, with the right-linearity restriction removed, are essentially least solutions to recursive equations using the operators `()` (the empty value), `l[T]` (singleton element), `S|T` (union), and `S,T` (sequencing). For example, the derived operator `T*` is defined by the equation:

```
X = T,X | ()
```

The XDuce implementation further allows attributes to be modeled as special *floating* elements of the form `@a`.

A regular expression type defines a set of XML values corresponding to all finite unfoldings. It is now a simple matter to build inductively an XML graph that defines the same set of XML values. The five operators are modeled by XML graphs as follows: `()` as an empty sequence node, `l[T]` as an element node, `S|T` as a choice node, `S,T` as a sequence node, and `@a` as an attribute node. In each case, a variable is modeled by an edge to the root node of the XML graph corresponding to its right-hand side.

Example The following regular expression type is represented as shown in Figure 8:

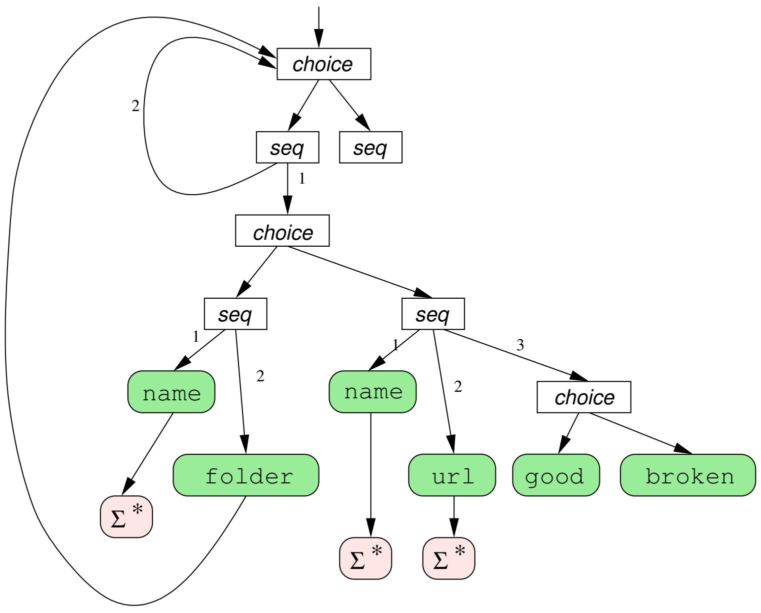


Figure 8: XML graph for a regular expression type.

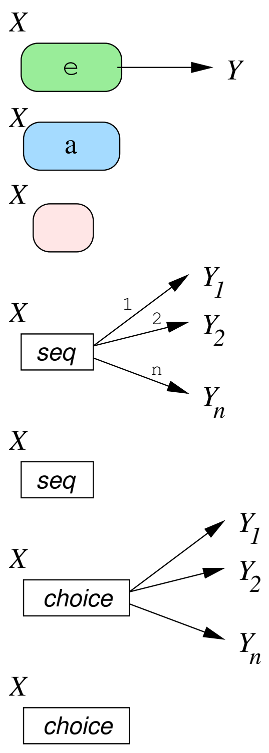
```
type Fld = Rcd*
type Rcd = name[String], folder[Fld]
         | name[String], url[String], Sts
type Sts = good[] | broken[]
```

The inverse translation is equally straightforward (ignoring gap nodes, interleave nodes, and regular string languages). Initially, we assign type variables to all element nodes, sequence nodes, and choice nodes. Then we define equations based on the outgoing edges, as shown in Figure 9. Finally, for the root nodes $R_1, \ldots, R_n$ we define the type equation $R = R_1 \mid \ldots \mid R_n$, and the type R is then the final result of the translation. Note that the resulting equations may violate the top-level recursion restriction of XDuce.

Any analysis that produces XML graphs is thus also able to infer (generalized) regular expression types. In this way, the work on XACT (see Section 6.1) can be viewed as an alternative to XDuce, supporting flow-sensitive type inference.

3.3. RELAX NG and XML Schema

The RELAX NG schema language [14] is based on regular tree grammars. A schema in this language consists of recursively defined *patterns* of various kinds, including the following: `element` matches one element with a given name



$$X = \texttt{e[}Y\texttt{]}$$

$$X = \texttt{@a}$$

$$X = \texttt{String}$$

$$X = Y_1\texttt{,}\ Y_2\texttt{,}\ \ldots\texttt{,}\ Y_n$$

$$X = \texttt{()}$$

$$X = Y_1\texttt{|}\ Y_2\texttt{|}\ \ldots\texttt{|}\ Y_n$$

$$X = X$$

Figure 9: From XML graphs to generalized regular expression types.

and with contents and attributes described by a sub-pattern; `attribute` similarly matches an attribute; `text` matches any character data or attribute value; `group`, `optional`, `zeroOrMore`, `oneOrMore`, and `choice` correspond to concatenation, zero or one occurrence, zero or more occurrences, one or more occurrences, and union, respectively; `empty` matches the empty sequence of nodes; and `notAllowed` corresponds to the empty language. In addition, the pattern `interleave` matches all possible mergings of the sequences that match its subpatterns.

Note that attributes are described in the same expressions as the content models. Still, attributes are considered unordered, as always in XML, and syntactic restrictions prevent an attribute name from occurring more than once in any element. Mixing attributes and contents in this way is useful for describing attribute–element constraints.

To ensure regularity (as in regular expression types), there is an important restriction on recursive pattern definitions: recursion is only allowed if passing through an `element` pattern.

Element and attribute names can be described with *name classes*, which can consist of lists of possible names and wildcards that match all names, potentially restricted to a certain namespace or excluding specific names.

To describe datatypes more precisely than with the `text` pattern, RELAX NG relies on an external language, usually the datatype part of XML Schema. Using the `data` pattern, such datatypes can be referred to, and datatype facets can be constrained by a parameter mechanism.

Although RELAX NG is an elegant and powerful schema language, W3C's XML Schema is a more widely used alternative. We choose to incorporate XML Schema via a subset of RELAX NG, as explained in detail below.

Restricted RELAX NG

We connect XML graphs and XML Schema using RELAX NG as a convenient intermediate language that avoids the many complicated technical details of XML Schema. However, we only use a subset of RELAX NG that is characterized as follows. This language, called Restricted RELAX NG, was first presented in [24]. It has been designed to achieve a compromise in expressiveness: (1) Its expressiveness is sufficient for making an exact and simple embedding of XML Schema, as explained below; (2) every Restricted RELAX NG schema can be converted into an XML graph, as also explained below; and (3) it makes XML graph validation (Section 4.2) more tractable than using XML Schema directly or supporting full RELAX NG.

First, we define some terminology.[1] We say that the *surface* of a pattern p is the set of sub-patterns of p, including p itself, where contents of `element` and `attribute` patterns are ignored. A pattern is a *content pattern* if its surface contains one or more `element`, `data`, or `text` patterns (or `list` or `value` patterns, which we otherwise ignore here for simplicity). A pattern is an *attribute list pattern* if its surface contains one or more `attribute` patterns.

A Restricted RELAX NG schema satisfies the following syntactic requirements:

[single-type grammar] For every `element` pattern p, any two `element` patterns that are in the surface of the child of p and have non-disjoint name classes must have the same (identical) content. (This requirement essentially limits the notation to single-type grammars [41], except that we retain attributes, datatypes, and name classes, and interleave constructs.)

[attribute context insensitivity] No attribute list pattern can be a `choice` pattern. Also, every `optional` attribute list pattern must have an `attribute` pattern as child. (This requirement prohibits context sensitive `attribute` patterns.)

[interleaved content] Every pattern that has a child whose surface contains an `interleave` content pattern must be a `group` or `element` pattern.

[1]Notice the similarity with the terminology used in the definition of XML graphs in Section 2.

Also, a `group` pattern whose surface contains an `interleave` content pattern must have only one content pattern child. (This requirement makes it easier to check inclusion of `interleave` patterns, as explained in Section 4.2.)

We here consider `ref` patterns as abbreviations of the patterns being referred to. For every `element` and `optional` pattern that has more than one child pattern, we treat the children as implicitly enclosed by a `group` pattern. Also, all `mixed` patterns are implicitly desugared to `interleave` patterns in the usual way.

From XML Schema to Restricted RELAX NG

Most XML Schema constructs map directly to RELAX NG, and we will not here explain the details of the translation. However, a few points are worth mentioning:

1. The `all` construct maps to the `interleave` pattern. Because of the limitations on the use of `all` in XML Schema, this does not violate the [interleaved content] requirement.
2. We can ignore default declarations since we only care about validation and not of normalization of the input—except that we treat an attribute or content model as optionally absent if a default is declared.
3. Wildcards can be converted into name classes. If `processContents` of an element wildcard is set to `skip`, then we make a recursive pattern that matches any XML tree.
4. The most tricky parts of the translation involve type derivations and substitution groups. Assume that an element e has type t and there exists a type t' that is derived by extension from t. In this case, an occurrence of e must match either t or t', and in the latter case e must have a special attribute `xsi:type` with the value t' (in the former case, the attribute is permitted but not required). We handle this situation by encoding the `xsi:type` information in the element name. More precisely, we create a new element pattern whose name is the name of e followed by the string `%`t' and whose content corresponds to the definition of t'. Each reference to e is then replaced by a choice between e and the variants with extended types. The `xsi:nil` feature is handled similarly. Now assume that another element f has type t' and is declared as in the substitution group of e. This means that f elements are permitted in place of e elements. In Restricted RELAX NG, this is expressed simply by replacing all references to e elements by choices of e and f elements. Again, because of limitations on the `all` construct and the substitution group mechanism in XML Schema, this cannot lead to violations of the [single-type grammar] requirement, nor of the general RELAX NG requirement that `interleave` branches must be disjoint.

Example Consider the following schema written in XML Schema:

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

  <element name="cardlist">
    <complexType>
      <sequence>
        <element ref="b:card"
                 minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>

  <element name="card" type="b:card_type"/>

  <complexType name="card_type">
    <sequence>
      <element name="name" type="string"/>
      <element name="email" type="string"
               maxOccurs="unbounded"/>
      <element name="phone" type="string"
               minOccurs="0"/>
    </sequence>
  </complexType>

</schema>
```

(This schema is also used in the Xact example in Section 6.1.) Assuming `cardlist` as root element name, this can be translated into the following Restricted RELAX NG schema, here using the compact RELAX NG syntax:

```
default namespace = "http://businesscard.org"

start = element cardlist { card* }
card = element card { card_type }
card_type = element name { xsd:string },
            element email { xsd:string }+,
            element phone { xsd:string }?
```

A *schema type* is the name of an element or a (simple or complex) type that is declared in a schema written in XML Schema. The *language*, $\mathcal{L}(t)$, of a schema type (or a RELAX NG pattern) t is defined as the set of XML documents or document fragments that are valid relative to the schema type (or pattern).

By the translation to Restricted RELAX NG, a schema type t corresponds to a pattern definition p_t. The translation from XML Schema to Restricted RELAX NG is exact in the sense that $\mathcal{L}(t) = \mathcal{L}(p_t)$. Moreover, the size of the output schema is proportional to the size of the input schema.

From Restricted RELAX NG to XML Graphs

Given a Restricted RELAX NG pattern p, an equivalent XML graph χ_p, always with precisely one root node, can be constructed quite easily by a recursive traversal. Again, the translation is exact, that is, $\mathcal{L}(\chi_p) = \mathcal{L}(p)$, and the size of the resulting XML graph is proportional to the size of the input schema.

First, we observe that every possible name class n defines a regular string language $\mathcal{L}(n)$. Namespaces are handled by expanding qualified names according to the applicable namespace declarations, as discussed in Section 2. Also, datatypes defined by `data`, `value`, `choice`, and `list` define regular string languages if using the XML Schema datatype library.[2]

Most pattern kinds have direct counterparts as nodes in XML graphs. Assume, for example, that p is an `element` pattern with name class n and contains a pattern c:

$p =$ `element` n `{` c `}`

In this case, the sub-pattern c is recursively translated into an XML graph with a root n_c. We then add a new element node n_p, which becomes the new root, and we set $\mathit{contents}(n_p) = n_c$ and $\mathit{strings}(n_p) = \mathcal{L}(n)$.

A datatype pattern can be transformed into a text node n where $\mathit{strings}(n)$ is set to the corresponding regular string language. The `notAllowed` pattern can be modeled as a choice node with no outgoing edges, and similarly, an `empty` pattern becomes a sequence node with no outgoing edges. A `zeroOrMore` pattern can be encoded by a choice node and a sequence node arranged in a cycle.

For `interleave` patterns, which are translated into interleave nodes, the [interleaved content] requirement in Restricted RELAX NG ensures that the requirement on interleave nodes in XML graphs, as defined in Section 2, is obeyed.

A `ref` pattern is handled simply by translating the named pattern being referred to. As a consequence, recursion in pattern definitions leads to cycles in the XML graph.

Example Translating the following pattern (here written using the compact RELAX NG syntax) results in the XML graph shown in Figure 1:

```
element ul { element li { xsd:integer { minInclusive="0" } }* }
```

4. Operations on XML Graphs

The various applications of XML graphs involve a number of interesting operations, several of which are of general interest. Others are more application

[2]In practice, we here ignore a few combinations of constraining facets and datatypes, such as the `value` facet together with the `float` datatype, which lead to unwieldy regular languages. These are uncommon cases that can be accommodated for without losing precision by slightly augmenting the definition of string edges.

specific, such as the operations in the XACT analyzer (Section 6.1) that model a range of basic operations on XML templates as transfer functions on XML graphs, or the operations in the XSLT analyzer (Section 6.4) for modeling XSLT instructions.

An example of an XML graph corresponding to an XML template, as used in XACT, is seen in Figure 5. XML documents are merely special cases of XML templates without gaps and can thus similarly be represented as XML graphs, as shown in Section 3.1. Additionally, as shown in Section 3.3, every schema written in XML Schema or Restricted RELAX NG can be converted into an equivalent XML graph.

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. As mentioned, XML graphs are also closed under language union but we have never encountered a need for performing this operation in practice.

4.1. Plugging

The plugging of gaps in XML templates is a central operation that must be modeled by a similar abstract operation on XML graphs. Figure 10 shows the case where a gap g in an XML template X is being plugged with an XML template Y (in XACT syntax this would be denoted by X.plug(g,Y)). Note that X and Y are represented as possibly overlapping nodes in the same XML graph, but with different sets of root nodes, variable edges, string maps, and gap maps. The resulting XML graph is obtained by using the roots of X, the union of the variable edges from X and Y, the union of the *strings* maps from X and Y, adding edges from the possibly open gap nodes named g in X to all the root nodes of Y, updating the status of the g gaps in X to CLOSED, and finally using the least upper bound of this *gaps* map with that from Y.

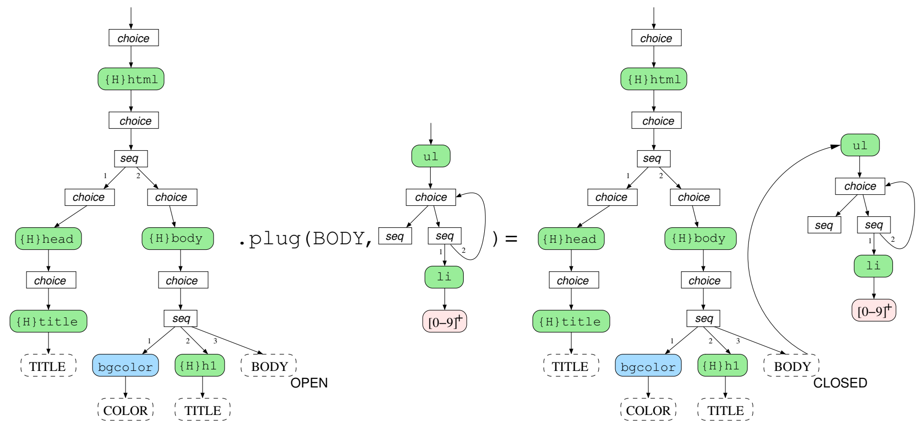


Figure 10: Example of abstract plug operation.

After any construction of a new XML graph, we perform a *sharpening* operation. For the parts of the XML graph that are unreachable from the root nodes we lower the information to $\perp$, that is, remove variable edges, reduce string sets to the empty language, and set the gap maps to $\emptyset$. This will clearly maintain a sound description but may improve precision through later abstract operations.

4.2. Validation

All our applications of XML graphs involve validation, that is, checking whether or not every XML document represented by a given XML graph χ is valid relative to a given Restricted RELAX NG pattern p:

$$\mathcal{L}(\chi) \subseteq \mathcal{L}(p)$$

Our algorithm, which was first described in [24], 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 the surfaces of nodes and the surfaces of patterns as context-free grammars and finite string automata over a common vocabulary and checking inclusion of their languages. This is explained in more detail in the following.

Given a pair of XML graph node n and a Restricted RELAX NG pattern p, we wish to determine whether $n \Rightarrow x\ ;\ t\ ;\ a$ implies $x \in \mathcal{L}(p)$ for every XML content x. We begin by considering the case where n is not an interleave node and p is not an `interleave` pattern.

1. First, a context-free grammar is constructed from the surface of n, considering element nodes and text nodes as terminals, choice/sequence/interleave nodes as nonterminals, and ignoring attribute nodes. Each text node terminal c is then replaced by a regular grammar equivalent to $\mathit{strings}(c)$. Thus, we have a context-free language L_n over the alphabet of element nodes and Unicode characters, describing the possible unfoldings of n (ignoring attributes). Similarly, p defines a regular string language L_p over `element` patterns and Unicode characters.
2. To obtain a common vocabulary, we now replace each element node n' in L_n by $\langle \mathit{strings}(n') \rangle$ (where $\langle$ and $\rangle$ are some otherwise unused symbols), and similarly for the `element` patterns in L_p.
3. Then we check that L_n is included in L_p with standard techniques for context-free and regular string languages [43]. If the check fails, a suitable validity error message is generated. For each pair (n', p') of an element node in L_n and an `element` pattern in L_p where $\mathit{strings}(n')$ and the name class of p' are non-disjoint, two checks are then performed:
   (a) We check that the attributes of n' match those of p': For each attribute node a in the surface if n', each name $x \in \mathit{strings}(a)$, and each value $y \in \mathit{strings}(a)$, a corresponding `attribute` pattern must occur in the surface of p'—that is, one where x is in the language of its name class and y is in the language of its sub-pattern; also, `attribute` patterns

occurring in p' that are not enclosed by `optional` patterns must correspond to one of the non-optional attribute nodes. Again, a suitable validity error message is generated if the check fails.

(b) We check recursively that *contents*(n') is valid relative to the sub-pattern of p'. Cycles in the XML graph and recursive definitions in the schema are handled coinductively using memoization.

(Note that correctness of this algorithm depends on the [single-type grammar] and [attribute context insensitivity] requirements in Restricted RELAX NG.)

For interleave nodes and `interleave` patterns, we exploit the restrictions on these constructs. Additionally, in RELAX NG, the sub-patterns of an `interleave` pattern must be disjoint (that is, no element name or `text` pattern occurs in more than one sub-pattern). Thus, if p is an `interleave` pattern, we simply test each sub-pattern in turn, projecting L_n onto the element names occurring in the sub-pattern, and then check that all element names occurring in L_n also occur in one of the sub-patterns.

This algorithm validates the XML graph χ relative to the schema pattern p in the sense that no error messages are generated if and only if $\mathcal{L}(\chi) \subseteq \mathcal{L}(p)$. The theoretical time complexity is determined by the size of the memoization table and the time it takes to fill one entry of the table. The table size is the number of nodes in χ times the number of sub-patterns in p. The work required for each table entry is worst case cubic time, but it depends only on L_n and L_p, which usually involve only small parts of the XML graph and the schema, respectively.

Compared to validation algorithms based on more expressive models [20, 23], this approach exploits, in particular, the single-type property of the schema to obtain a simpler algorithm and to provide more informative error messages.

As an interesting side-effect of our approach, we get an inclusion checker for Restricted RELAX NG and hence also for XML Schema and DTD: Given two schema types, t_1 and t_2, convert t_1 to an XML graph χ_{t_1} using the algorithm described in Section 3.3 and then apply the validation algorithm above on χ_{t_1} and t_2. Validation succeeds without reporting errors if and only if $\mathcal{L}(t_1) \subseteq \mathcal{L}(t_2)$.

Example Assume that we wish to validate the XML graph shown in Figure 1 relative to the pattern p defined by

$p =$ `element ul { element li { xsd:decimal }* }`?

We first produce the context-free grammar for the root node e_1:

$N_{e_1} \rightarrow \langle\texttt{ul}\rangle$

The regular language for p is $(\langle\texttt{ul}\rangle)^?$, and clearly inclusion holds: $L_{e_1} \subseteq L_p$. We then proceed by constructing the context-free grammar for the content c of e_1:

$N_c \rightarrow N_{s_1} \mid N_{s_2}$
$N_{s_1} \rightarrow \epsilon$
$N_{s_2} \rightarrow \langle\texttt{li}\rangle \mid N_c$

The regular language for the contents of the corresponding `ul` element pattern is $(\langle \texttt{li} \rangle)^*$, and again inclusion holds. Continuing recursively, we find out that the language of node t is included in the language of the sub-pattern `xsd:decimal`, so we conclude that all documents represented by the XML graph are in fact valid relative to the schema.

4.3. 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 selected by p in $\mathcal{L}(\chi)$, starting from a root node.

Each node being selected in an XML graph may correspond to many actual nodes in the possible unfoldings and the results of XPath predicates may depend on the actual values in unfoldings. Our algorithm supplies detailed answers that capture the status of a given node with some precision. Each node $n \in \mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{A} \cup \mathcal{N}_\mathcal{T}$ is mapped to one of the following values:

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;

NEVER: the conditions for ALL and NONE are both satisfied, that is, in every unfolding, no tree node corresponds to n; and

DONTKNOW: none of the above can be determined.

These six values form a partial order:

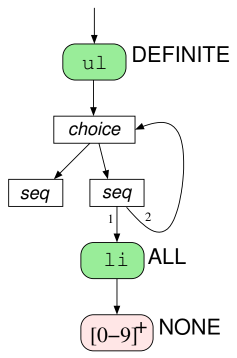
$$\begin{tikzcd}
 & & \mathsf{DONTKNOW} \arrow[dll, dash] \arrow[d, dash] \arrow[drr, dash] & & \\
\mathsf{NONE} \arrow[dr, dash] & & \mathsf{ALL} \arrow[dl, dash] \arrow[dr, dash] & & \mathsf{SOME} \arrow[dl, dash] \\
 & \mathsf{NEVER} & & \mathsf{DEFINITE} &
\end{tikzcd}$$

and the algorithm seeks to provide answers as low as possible, which corresponds to increased precision.

Only a subset of the full XPath language is handled. We currently restrict ourselves to the downwards `child`, `descendant-or-self`, and `attribute` axes. Also, only the predicates that are nested location paths are considered in the analysis (other expressions correspond to a nondeterministic choice). If required, the full collection of XPath axes could be handled by a conservative rewriting into downwards axes as explained in [38].

The algorithm evaluates an XPath expression one location step at a time, first modeling the axis, then the node test, and finally the predicate, as explained in detail in the article [26].

Example The result of evaluating the XPath expression `//*` on the XML graph from Figure 1 is as follows:



The `ul` node is part of every unfolding and is always selected by the expression. The `li` node may be absent in some unfoldings (actually just one) but when present it is always selected. The text node is present in some unfoldings but is never selected.

This abstract evaluation of XPath expressions is particularly useful when analyzing Xact programs. However, it can also be used for extending the XML graph validator to check element prohibitions (for example, that `form` elements cannot be nested in XHTML).

5. Implementation

The Java library `dk.brics.schematools` [27], 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 into 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 [22] supports conversion from XML Schema to RELAX NG but has several deficiencies; Trang [10] 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, as explained in Section 4.2.

6. 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, 8, 26, 24, 6, 38, 3, 25, 37]. They all use different aspects of the package and the languages they consider span a wide spectrum. Despite their differences, these fully automated applications follow 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 depend on the application domain. Third, the resulting XML graphs are analyzed, generally using the XML graph validation tool. The XML graph library is a key component in all these applications.

6.1. Xact

The Xact language extends Java with domain-specific support for manipulating XML documents [26, 24]. 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 decomposed 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 [39].

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 XML variable `wrapper` is by the method `setDefaultWrapper` initialized to contain the skeleton for an XHTML document with white background. The program then reads a collection of business cards from the file `cards.xml` that is declared to conform to the type `cardlist` from the XML schema presented in Section 3.3. This collection is transformed by the `transform` method, which invokes the `makeList` method and plugs the result into the wrapper. The `makeList` method uses an XPath expression to iterate through those `card` elements that contain a `phone` element as a child. For each of those cards, the name and phone number are selected and plugged into an `li` element, which is then accumulated into an XML variable that initially contains an empty `ul` element.

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

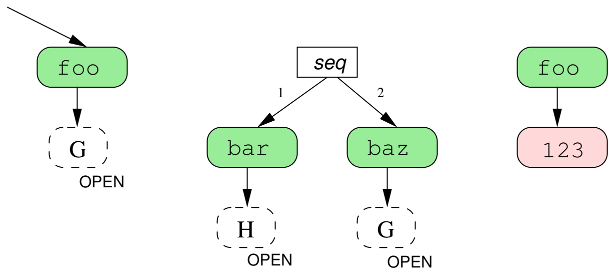
Since XACT is an extension of full Java, the analysis must first construct an ordinary flow graph for the Java program. This is done using the Soot framework [46]. Subsequently, we perform a standard dataflow analysis [21] but with the highly specialized lattice structure of XML graphs described in Section 2. 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 4.

EXAMPLE Consider the following fragment of an XACT program:

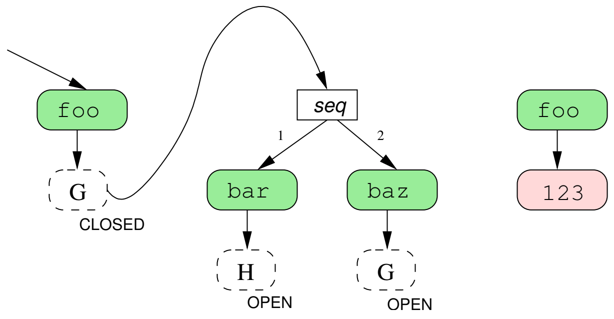
```
x = [[<foo><[G]></foo>]];
y = x.plug("G", [[<bar><[H]></bar><baz><[G]></baz>]];
if (z) y = y.plug("G", [[<foo>123</foo>]]);
```

When this program is analyzed, every XML value is described by an XML graph that contains the nodes corresponding to all the template constant that are used. Thus, all XML graphs used in the analysis of a given program are compatible. After the first line, the value of the variable `x` is described by the following XML graph:

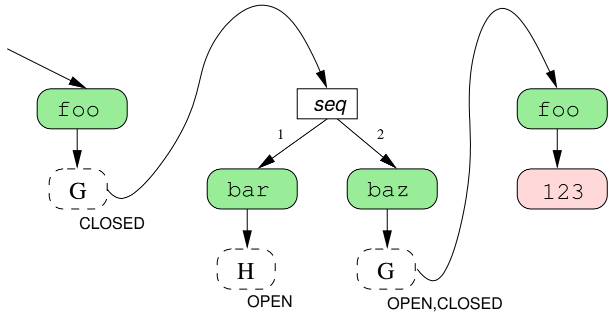


Every occurrence of an XML node in templates is represented as an individual node in the XML graph. This means that the analysis has a notion of polyvariance at the level of term constructors. The relevant gap information is here shown as extra labels on the nodes; for example, $n \in \mathit{open}(\mathrm{H})$ where n is the node with the H gap, and implicitly $\mathit{egaps}(\mathrm{G}) = \{\mathsf{OPEN}\}$ because of the leftmost G gap node. Irrelevant parts of an XML graph are simply unreachable from the roots.

After the second line of the program, the value of the variable `y` is described by the following compatible XML graph (using the definitions from Section 4.1):



The third line describes the variable `y` as a merge of two XML graphs: one where `z` is true and the plug operation is performed, and one where it is not performed. The resulting XML graph looks as follows:



Note that since the leftmost `G` gap under `foo` is closed, it is not involved in the potential plug operation. The fact that the plug operation may not have been performed in the merged result is modeled by the other `G` gap being both open and closed.

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 4. Also, for plug operations it can be 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.

6.2. Java Servlets

The Xact project introduces a novel extension of Java for manipulation of XML templates. In contrast, the Java Servlets framework works 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>");
  }
}
```

The `Wrapper` class is responsible for printing headers and footers that define a common skeleton of all XHTML documents. The program contains two servlets. The `Entry` servlet presents the user with an XHTML form, collects the name of a person, which is then submitted to the second servlet. The `Show` servlet reads the name as a parameter, retrieves phone information from an external databases, and presents the result.

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 [25], a program analysis that attacks these problems is presented, based on XML graphs. The paper [37] 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 [9]. 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 [28] and grammar approximations by Mohri and Nederhof [36]. 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 11.

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

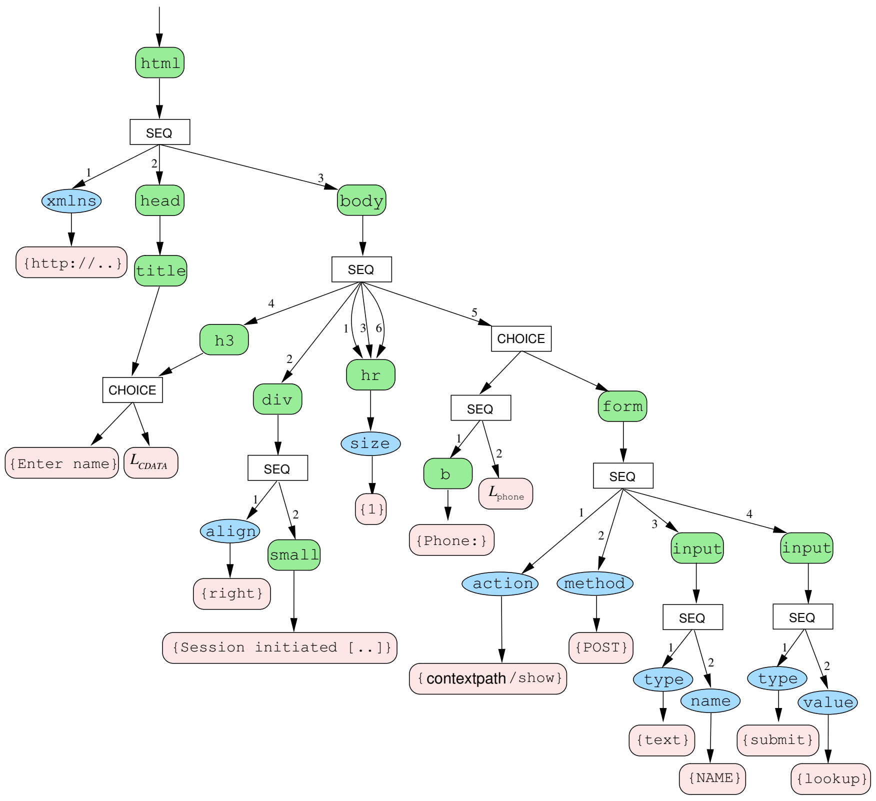


Figure 11: XML graph for servlet example.

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.

6.3. 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 [14] that has an alternative, compact, non-XML syntax [12] (as we have used in the example schemas shown in previous sections). Other languages with dual syntax include BibTeXML [18] and the Wiki notation [30]. 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 first line declares the namespace associated with the empty prefix. The next four lines define some *regular expressions*, which are used for describing syntactic tokens. For example, `Name` matches one or more blocks of alphabetic characters, separated by space characters. The remaining lines define *grammar productions* where each nonterminal, such as `person`, has two right-hand sides. The first (following the `:` symbol) uses alternative syntax, while the second (following the subsequent `=` symbol) uses XML syntax. The two right-hand sides must, however, use the same named nonterminals and tokens, which enables an inductive translation to take place.

The XSugar tool analyzes the grammar to ensure reversibility of this translation between the two versions, which involves an approximate decision procedure for ambiguity of grammars [4]. The remaining problem, which is relevant for this article, 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:

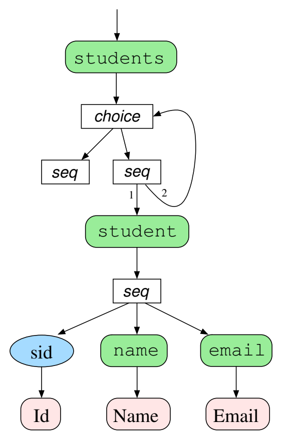


Figure 12: XML graph for XSugar example.

- each nonterminal becomes a choice node with a child for each of its productions;
- 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 12.

XML Graph Analysis

Static validation of the XSugar program is simply obtained by means of the main algorithm from Section 4. 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.

6.4. XSLT

An interesting challenge in the area of static validation is posed by XSLT stylesheets [11]: 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 [38] and generalized to XSLT 2.0 in [29]. Earlier work in this area provided partial solutions [2, 45, 16] or looked at idealized languages [35, 32, 33, 31]. 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>

  <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>
```

The root element of the registration document is matched by the first template which generates an XHTML wrapper and uses `apply-templates` to process all child nodes as item of an enclosing ordered list. Ordinary `name` nodes are handles by the template with pattern `*`, which generates its contents as simple list items. The `group` nodes are handled by the specific template which generates a tiny table with the affiliation and a ® symbol if it is a private company. The group members are through an `apply-templates` instruction, which also passes

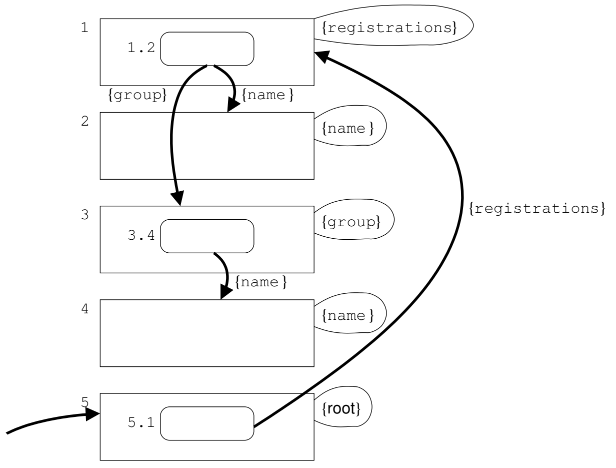
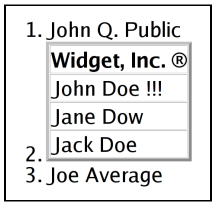


Figure 13: Flow graph for XSLT example.

the identity of the group leader as a parameter, handled by a special template that lists them as table rows and adorns the name of the leader with triple exclamation marks. For the above example document, the resulting XHTML document is rendered as follows by a standard browser:



1. John Q. Public
2. **Widget, Inc. ®**
   John Doe !!!
   Jane Dow
   Jack Doe
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

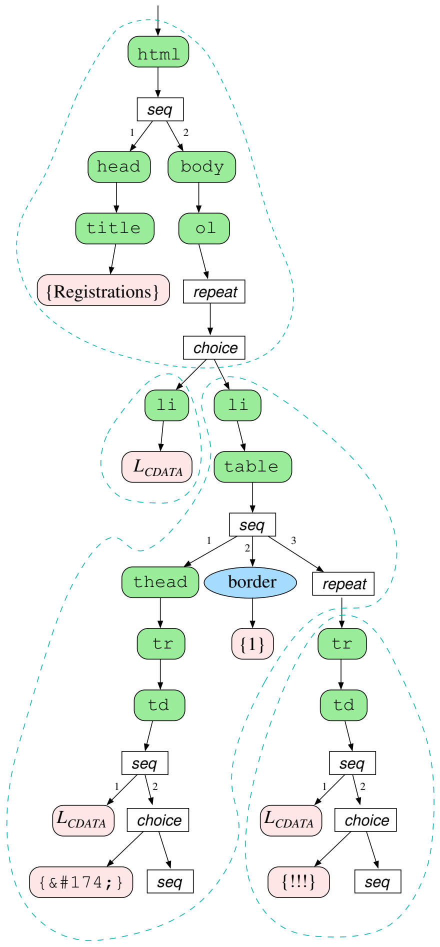


Figure 14: XML graph for XSLT example.

187,015 lines of code written by hundreds of different authors. For our example stylesheet, the flow information is summarized as shown in Figure 13.

Based on this flow graph, the details of the stylesheet and the input schema, our algorithm constructs an XML graph that describes 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 14. The *repeat* abbreviates the choice–sequence cycle 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 validation algorithm from Section 4. 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.

7. 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, XSugar, and XSLT. The implementation is available in an open source software package.

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. In *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, volume 4783 of *LNCS*. Springer-Verlag, July 2007.

[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. *Information Systems*, 33(4), June 2008. Earlier version in Proc. 10th International Workshop on Database Programming Languages, DBPL '05, Springer-Verlag, LNCS vol. 3774.

[7] 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.

[8] 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.

[9] 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.

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

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

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

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

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

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

[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 and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proc. 3rd International Workshop on the World Wide Web and Databases, WebDB '00*, volume 1997 of *LNCS*. Springer-Verlag, May 2000.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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. Full version available as BRICS RS-06-10.

[26] 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.

[27] Christian Kirkegaard and Anders Møller. dk.brics.schematools, 2007. `http://www.brics.dk/schematools/`.

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

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

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

[31] 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.

[32] 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.

[33] 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.

[34] Laurent Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4):290–311, 2000.

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

[36] 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.

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

[38] Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. *ACM Transactions on Programming Languages and Systems*, 29(4), July 2007.

[39] 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.

[40] Anders Møller and Michael I. Schwartzbach. XML graphs in program analysis (invited paper). In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '07*, January 2007.

[41] 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.

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

[43] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.

[44] 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/`.

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

[46] 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.