Static Analysis of XML Transformations in Java

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

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

We introduce XACT, a high-level approach for Java using XML templates as a first-class data type with operations for manipulating XML values based on XPath. In addition to an efficient runtime representation, the data type permits static type checking using DTD schemas as types. By specifying schemas for the input and output of a program, our analysis algorithm will statically verify that valid input data is always transformed into valid output data and that the operations are used consistently.

***Index Terms*— D.3.3 Language Constructs and Features, I.7.2.f Markup Languages, D.2.1 Requirements/Specifications**

I. Introduction

Extensible Markup Language, XML [1], has since its introduction in 1998 gained considerable interest from industry and now plays an important role in the exchange of a wide variety of data on the Web. Although XML, technically, is merely a linear syntax for ordered labeled tree structures, it has proven useful as a notation for structuring information in general.

The syntax of an XML-based language is specified using a vocabulary of elements and attributes together with rules for constraining their use. There exists a variety of schema languages, such as DTD [1], XML Schema [2], or DSD2 [3], allowing the syntax to be formalized. An XML document is *valid* relative to a given schema if all the syntactic requirements specified by the schema are satisfied in the document. The *language* $\mathcal{L}(S)$ of a schema S is the set of XML documents that are valid relative to S.

A popular XML-based language is XHTML [4], the "XML-ized" variant of HTML. The XHTML language is widely used in *interactive* Web services where the clients are human beings that use browsers to interact with the servers. A recent trend is to move from interactive Web services towards *application-to-application* Web services, where the clients are not humans with browsers but general programs. This calls for specialized XML-based languages to mediate communication between clients and servers. As an example, Amazon.com now provides an XML interface [5] that allows other programs to search for product information. These other programs may combine that information with data from other sources, extract relevant parts and, for example, transform the results into other XML documents to interact with yet another group of programs.

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

*Corresponding author. BRICS, Department of Computer Science, University of Aarhus, Denmark. Email: `amoeller@brics.dk`

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

Existing general-purpose programming languages do not provide any special support for XML transformations. With these languages, the programmer may choose to model XML data either 1) as text strings, or 2) as DOM [6] tree structures (or variants of that, such as JDOM [7]). The first approach is often used for languages as XHTML where documents are being constructed but rarely deconstructed, whereas the second is more used for languages and transformation that involve both construction and deconstruction of documents. We shall argue that both approaches are low-level in the sense that they are often error-prone and tedious to use.

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

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

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

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

We work in the context of JWIG [8], [9], an extension of Java that, among other features, provides a mechanism for construction of XML documents using *XML templates* and *plug operations*, which we briefly recapitulate in Section II. Our previous results included a static analysis for checking that the constructed documents are always valid relative to a given DSD2 schema. However, the mechanism only supported *construction* of XML documents, not *deconstruction*. This has shown to be sufficient for interactive Web services that dynamically create XHTML documents, but, as explained earlier, application-to-application Web services require general XML transformations, which also includes deconstruction. Furthermore, the previous results were obtained under the assumption that XML documents are built from a set of constant XML templates. This is also a valid assumption for interactive Web services, but not for application-to-application Web services, where the constituents of the result of an XML transformation are often input from other Web services. In the present article we generalize the previous results to general XML transformations that also involve deconstruction and importing of XML templates.

Contributions

Our contributions in this article are the following:

- A novel data type with high-level operations for defining XML transformations in Java;
- a static analysis technique based on a notion of summary graphs;
- an algorithm for symbolic evaluation of XPath expressions [10] on summary graphs, which is essential in the static analysis to model XPath operations that select fragments of XML values;
- an algorithm for converting DTD schemas into summary graphs, which is used in the static analysis for modeling type cast operations;
- experimental evidence that the approach is practically feasible; and
- a survey of existing techniques for defining XML transformations.

Preliminary results were described in [11]. In a companion paper [12], we show that our data type also permits an efficient runtime representation. Although we focus on Java, our ideas can be applied to other general-purpose high-level programming languages since we do not depend on any Java-specific language constructs.

Overview

Section II explains our approach, named XACT. It involves DTD and XPath for expressing classes of XML values and selecting fragments of individual values. The operations in XACT can be performed efficiently with a suitable runtime representation, which we mention briefly and describe in detail in a separate paper. In Section III, we describe *summary graphs*, a formalism that provides the foundation for the static program analysis, which we describe in Section IV. This analysis encompasses techniques for symbolically evaluating XPath expressions on summary graphs and converting DTD schemas into summary graphs. Our prototype implementation and a number of benchmark tests of the analyzer are described in Section V.

Appendix I contains a comprehensive survey of related work on language support for XML transformations. In Appendix II, we show how the basic XACT operations can be extended with convenient syntactic sugar.

II. XML OPERATIONS USING DTD AND XPATH

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

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

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

By building on an imperative language, our mechanism is operational and in that respect closer to, for example, JDOM, than to a declarative language as XQuery. However, an important design choice is that our data type for XML templates is *immutable* [13]. There are several reasons for this choice: As in pure functional languages, having no side-effects often permits a cleaner programming style. For example, there is no need for explicit copying of values, thread safety comes for free, and the use of value factories is possible. Furthermore, since side-effects can be difficult to control, having immutable data avoids a significant class of programming errors. Finally, the crucial point in our situation is that immutability is a necessity for development of a feasible program analysis. It would not be possible to transfer our program analysis techniques to a mutable data type as, e.g., JDOM.

We represent XML values as *XML templates* in the style of JWIG [8]. An XML template is a wellformed XML fragment that may contain named *gaps* where other templates or strings may be inserted. The gaps may appear in place of elements or attribute values. In JWIG, this has proven to constitute an intuitive and flexible mechanism for XML document construction.

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

$$\begin{array}{rcll}
xml & : & str & \text{(character data)} \\
& | & \texttt{<}name\ atts\texttt{>}\ xml\ \texttt{</}name\texttt{>} & \text{(element)} \\
& | & \texttt{<[}g\texttt{]>} & \text{(template gap)} \\
& | & xml\ xml & \\
atts & : & name\texttt{="}str\texttt{"} & \text{(attribute constant)} \\
& | & name\texttt{=[}g\texttt{]} & \text{(attribute gap)} \\
& | & atts\ atts & \\
& | & \varepsilon &
\end{array}$$

Here, str denotes an arbitrary Unicode string, $name$ is an identifier, and g is a gap name. Actual XML values must of course be further constrained to be wellformed according to the XML 1.0 specification [1]. Empty elements may be written in the usual alternative notation <$name\ atts$/>. Moreover, in this description we abstract away all inlined DTD information, comments, and processing instructions.

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

DTD

The DTD formalism is a simple schema language for XML and is described in the XML specification [1]. A DTD schema is a grammar for a class of XML documents defining for each element the required and permitted child elements and attributes. The *contents* of an element are the sequence of its immediate children. It is specified using a restricted form of regular expressions over element names and `#PCDATA`, which refers to arbitrary character data. Attributes can be declared as required or optional for a given element, and their values can be constrained to finite collections of fixed strings. We ignore the special attribute types `ID`, `IDREF`, `ENTITY`, etc. We consider a DTD schema as a specification of an *XML type* in XACT.

The following example, which we use in later examples, is a DTD schema for collections of recipes:

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

This data model support both *simple* ingredients, consisting of a `name` and possibly an `amount` and a `unit`, and *composite* ingredients, which are described recursively by sub-recipes.

The JWIG validity analysis described in [8] uses a more powerful schema language, DSD2 [3], which is capable of capturing more complex syntactic requirements than DTD. The main reason for using DTD here is that our generalization of the XML cast operation, as explained in the following sections, requires translation from schemas into our summary graphs, which can be done straightforwardly and precisely for DTD.

XPath

XPath [10] is a simple but versatile DSL for addressing elements, attribute values, and character data—generally called *nodes*—in XML documents. It has proven powerful as a sub-language, for example in XSLT, for locating document fragments and as a pattern matching mechanism.

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

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

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

In our application of XPath, we restrict ourselves to the `child`, `descendant-or-self`, and `attribute` axes. This means that all evaluation is top-down, which is sufficient for all the transformations we mention in Section V and simplifies both the runtime system and the analyzer. (If the need for other axes should arise, it is trivial to support all axes in the runtime system, and the analysis could be extended correspondingly with a manageable loss of precision as consequence.) A similar approach is taken in the *fxt* language [14]. Conveniently, XPath offers some syntactic sugar for these axes: `child` is the default axis, `/descendant-or-self::node()/` may be written as `//`, and `attribute` may be written as `@`. The example above may then be abbreviated as follows:

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

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

An XPath expression is evaluated relative to an XML template using an implicit template root node as context node, similarly to the root node in the XPath data model.

Basic XML Operations

The class `XML`, which represents XML templates, allows several operations that are shown in Figure 1. The class is immutable: all operations return new values without changing the incoming values.

The `constant` operation constructs an XML template from the syntax generated by the *xml* nonterminal in the previously described grammar; the `toString` operation translates in the opposite direction. The argument to `constant` must be a constant. The `equals` operation determines equality of two templates, and `hashCode` returns a consistent hash code.

The `plug` operation is used to insert values into the specified gaps in a template. The operation is defined in four variants accepting strings, templates, or arrays of these. In the array versions, all occurrences of the named gap are plugged in document order with the values occurring in the array. If the lengths do not match, then superfluous array values are ignored and remaining gaps are plugged with the empty string. For the case where an element contains multiple attribute gaps, these are ordered lexicographically by attribute name. In the non-array version, all occurrences of the named gap are plugged

`static XML constant(String` s`)`	– creates an XML template from a constant string
`String toString()`	– converts this XML template into its textual representation
`boolean equals(Object` o`)`	– determines equality of this template and o
`int hashCode()`	– returns the hash code of this template
`XML plug(Gap` g`, XML` x`)`	– inserts a copy of x into all g gaps in a copy of this template
`XML plug(Gap` g`, String` s`)`	– as the previous, but for a string
`XML plug(Gap` g`, XML[]` xs`)`	– inserts the templates in xs into the g gaps in a copy of this template
`XML plug(Gap` g`, String[]` ss`)`	– as the previous, but for a string array
`XML[] select(XPath` p`)`	– returns all sub-templates selected by p
`XML gapify(XPath` p`, Gap` g`)`	– converts all sub-templates selected by p into g gaps
`XML close()`	– removes all open template gaps and all attributes with open gaps
`static XML[] group(XML[]` xs`, XPath` p`)`	– groups the templates in xs according to p
`XML cast(DTD` d`)`	– throws a runtime exception if this template is invalid relative to d
`static XML get(String` s`, DTD` d`)`	– converts s into a template and checks validity relative to d
`XML analyze(DTD` d`)`	– instructs the analyzer to verify that this template is valid relative to d

Fig. 1. Methods in the `XML` interface for performing basic XML template operations.

with the given value. Attempts to plug templates into attribute gaps will result in runtime errors. A gap that has not been plugged is said to be *open*.

As an example, plugging the single template

```
<ingredient name="salt" amount=[x] unit="teaspoon"/>
<[ingredients]>
```

into the `ingredients` gap of the template

```
<recipe><[title]>
<[ingredients]><[preparation]></recipe>
```

yields the following template:

```
<recipe><[title]>
<ingredient name="salt" amount=[x] unit="teaspoon"/>
<[ingredients]><[preparation]></recipe>
```

The `select` and `gapify` operations first find the node set indicated by the XPath expression using an implicit root node as initial evaluation context. In `select`, the subtrees rooted by nodes in this set are then copied in document order to form the resulting template array. Attribute gaps in the node set are ignored, and for normal attributes, their values are converted into character data. In `gapify`, the selected nodes and their sub-trees are each replaced by a gap with the given name; however, if one selected node is an ancestor of another, then only the ancestor is considered. The `close` operation closes all gaps in a template by removing template gaps and for each attribute gap, the entire attribute is removed.

To exemplify the `gapify` operation, if the template produced by the plug operation above is subjected to a gapify operation with gap name `first` and XPath expression `recipe/ingredient`, the result is the following:

```
<recipe><[title]>
<[first]>
<[ingredients]><[preparation]></recipe>
```

The `group` operation groups an array of templates according to a criterion specified by an XPath expression: for each template, the XPath expression is evaluated, and all templates where the evaluation gives the same result are merged in the order of occurrence. As an example, grouping the array consisting of the following three templates

```
<city name="Aarhus" country="Denmark" pop="223" />

<city name="New York" country="USA" pop="19,000" />

<city name="Copenhagen" country="Denmark" pop="1,084" />
```

according to the expression `city/@country` yields the following two templates:

```
<city name="Aarhus" country="Denmark" pop="223" />
<city name="Copenhagen" country="Denmark" pop="1,084" />

<city name="New York" country="USA" pop="19,000" />
```

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

All arguments of types `Gap`, `XPath`, and `DTD` are required to be constant. However, variables are permitted in the XPath expressions: all program variables that have a primitive type and whose declaration scope covers the XPath expression can be used.

Note that, e.g., most JDOM operations trivially are special cases of these operations – except that our data type is immutable, as explained earlier. The `parent` operation in JDOM does not have a counterpart in XACT since we always refer to the roots of the XML templates.

An XML transformation typically has the following form:

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

where input and output XML is represented textually. As a simple example, consider the following method that sorts the recipes in a given collection:

```
String sort(String s) {
  XML c = XML.get(s, DTD.make("file:recipes.dtd"));
  XML[] r = c.select("/collection/recipe");
  Arrays.sort(r, new RecipeComparator());
  c = c.gapify("/collection/recipe","g").plug("g",r);
  c.analyze(DTD.make("file:recipes.dtd"));
  return c.close().toString();
}
```

where the criterion for sorting recipes is lexicographic order of the titles:

```
public class RecipeComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    XML x1 = ((XML)o1), x2 = ((XML)o2;
    String s1 = x1.select("//title/text()")[0].toString();
    String s2 = x2.select("//title/text()")[0].toString();
    return s1.compare(s2);
  }
}
```

In Appendix II we show a number of extra operations that can be added as syntactic sugar on top of the basic XACT operations. For example, we allow XML constants and XPath expressions to be written directly in the usual XML and XPath syntax instead of as Java strings.

The program analysis described later will at compile time check that 1) each `analyze` operation is valid in the sense that the given template at runtime is guaranteed to be valid relative to the DTD schema, and 2) each `plug` operation always succeeds, that is, templates are never plugged into attribute gaps. Furthermore, if the analysis detects that an XPath expression in a `select`, `gapify`, or `group` operation will never select any nodes, or that a `plug` operation never has any effect because the specified gap is never present, then a warning is issued.

Runtime Representation

We show in a separate paper [12] that our data type for XML templates permits an efficient runtime representation, despite being immutable. We use a lazy non-copying data structure in which operations are merely noted to have happened until their effects are required to be observed. We obtain nearly optimal asymptotic complexities of the basic operations, since `plug` and individual moves from a parent node to its first child and from a node to its next sibling happen in amortized almost constant time. The `toString` operation is performed in linear time in the size of the resulting string. The complexity of `select` and `gapify` is bounded by the evaluation time for the associated XPath expression. The time for performing a `group` operation is bounded by the time for evaluating the XPath expression on each array entry and comparing the results. The `analyze` operation has no effect at runtime. The `cast` and `get` operations perform a linear time DTD validation. We also maintain a Java `hashCode` for XML objects and thus support a full `equals` method in constant time in the negative case and in amortized linear time in the positive case. All this assumes that we avoid a pathological case where templates containing only gaps are nested to an unbounded depth. We expect that a tuned implementation will be comparable to the runtime performances of dedicated tools such as JDOM and XSLT.

III. SUMMARY GRAPHS

To obtain static guarantees, we apply the standard dataflow analysis framework [15], [16]. This involves three steps: 1) obtaining an abstract *control-flow graph* for the given program; 2) defining a *lattice* modeling the abstract data that the analysis manipulates; and 3) describing all operations in the control-flow graph in terms of *transfer functions* that operate on the lattice values.

The construction of control-flow graphs from Java programs is described in detail in [8]. We use a different family of statements here, but the overall approach is the same and we do not describe it further—however, we note that arrays are modeled by merging their entries using weak updating. Our lattice is a variant of the *summary graph* lattice defined in [8] – we here use a notion of *normalized* summary graphs, as defined below. We need to modify the definition to accommodate the modeling of XPath expressions that may address individual nodes in XML fragments. The transfer functions are described in Section IV.

Given a program and all DTD schemas it refers to in `cast` and `get` operations, we fix a number of sets and functions to be used by all summary graphs that occur during the analysis: The sets E, A, and G contain the element names, attribute names, and gap names, respectively, that occur in the program and in the schemas. Let $N_\mathcal{E}$, $N_\mathcal{A}$, $N_\mathcal{C}$, and $N_\mathcal{T}$ be finite disjoint sets of element, attribute, chardata, and template nodes, respectively. Intuitively, the former three sets represent the possible elements, attributes, and chardata sequences, respectively, that may arise when running the program. The template nodes represent sequences of template gaps, which either occur explicitly in template constants or implicitly due to XACT operations or DTD schemas. More precisely,

- $N_\mathcal{E}$ contains a node for each occurrence of an element in a template constant in the program and one for each element description in the schemas. The function $name : N_\mathcal{E} \to E$ returns the corresponding element name.
- $N_\mathcal{A}$ contains a node for each occurrence of an attribute in a template constant and one for each attribute description in the schemas. The function $name : N_\mathcal{A} \to A$ returns the corresponding attribute name. Each element node is associated a set of attribute nodes, $attr : N_\mathcal{E} \to 2^{N_\mathcal{A}}$ corresponding to the element attributes.
- $N_\mathcal{C}$ contains a node for each maximal chardata sequence in a template constant and one for each occurrence of `plug`, `select`, and `#PCDATA`.
- $N_\mathcal{T}$ contains a node for each node in $N_\mathcal{E}$, one for each template constant, one for each occurrence of `select`, `group`, or `gapify`, and one for each sub-expression of the content model descriptors in the schemas. Each element node is associated a template node, $contents : N_\mathcal{E} \to N_\mathcal{T}$, corresponding to the element contents. Each template node has a sequence of gaps, $gaps : N_\mathcal{T} \to G^*$, which we define in Section IV.

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

A *summary graph* SG is a structure:

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

where

$$\frac{\begin{array}{c} n \in N_{\mathcal{E}} \quad SG \vdash \mathit{contents}(n) \Rightarrow d \quad \mathit{name}(n) = e \\ \mathit{attr}(n) = \{a_1, \ldots, a_k\} \quad SG \vdash a_i \Rightarrow b_i \quad \text{for all } i = 1, \ldots, k \end{array}}{SG \vdash n \Rightarrow \texttt{<}e\ b_1 \ldots b_k\texttt{>}\ d\ \texttt{</}e\texttt{>}} \qquad \frac{n \in N_{\mathcal{C}} \quad s \in S(n)}{SG \vdash n \Rightarrow s}$$

$$\frac{n \in N_{\mathcal{A}} \quad \mathit{name}(n) = a \quad s \in S(n)}{SG \vdash n \Rightarrow a\texttt{="}s\texttt{"}} \qquad \frac{n \in N_{\mathcal{A}} \quad \mathit{name}(n) = a \quad n \in \mathit{open}(P(g))}{SG \vdash n \Rightarrow a\texttt{=[}g\texttt{]}} \qquad \frac{n \in N_{\mathcal{A}} \quad n \in \mathit{removed}(P(g))}{SG \vdash n \Rightarrow \epsilon}$$

$$\frac{n \in N_{\mathcal{T}} \quad \mathit{gaps}(n) = g_1 \ldots g_k \quad SG, g_i \vdash n \Rightarrow d_i \quad \text{for all } i = 1, \ldots, k}{SG \vdash n \Rightarrow d_1 \ldots d_k}$$

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

Fig. 2. Inference rules for unfolding of summary graphs.

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

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

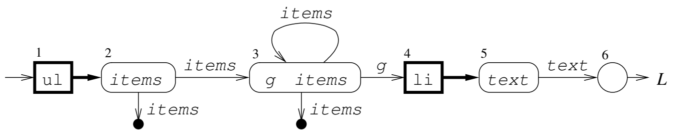
Intuitively, the language $\mathcal{L}(SG)$ of a summary graph SG is the set of XML templates that can be obtained by "unfolding" it, starting from a root node and plugging elements, templates, and strings into gaps according to the edges. A template edge $(n_1, g, n_2) \in T$ informally means that n_2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

We need the gap presence map to determine where edges should be added when modeling `plug` operations, to model the removal of gaps with the `close` operation, to detect when `plug` operations may fail because the specified gaps are not open, and to model and check XPath evaluations. Given that $P(g) = (p_1, p_2, p_3, p_4)$, let $\mathit{open}(P(g)) = p_1$, $\mathit{removed}(P(g)) = p_2$, $\mathit{tgaps}(P(g)) = p_3$, $\mathit{agaps}(P(g)) = p_4$. Informally, the open and $\mathit{removed}$ components specify which nodes may contain open or removed g gaps, and tgaps and agaps describe the presence of template gaps and attribute gaps, respectively. The value $\{\mathsf{OPEN}\}$ 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. ($\emptyset$ never occurs here.)

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

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

This can be illustrated as follows:



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

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

The unfolding of summary graphs can be formalized as

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

where the *unfolding relation*, $\Rightarrow$, is defined by induction in the structure of the summary graph according to Figure 2, considering only finite terms. The first six rules define how a node may be unfolded according to the different kinds of nodes: For element nodes, we look up the element name, attributes, and contents, and unfold attributes and contents recursively. For character data nodes, we look up the possible values in the string edges. For attribute nodes, there are three rules: one unfolds according to the string edges, one checks whether the attribute gap may be open according to the gap presence map, and one checks whether the attribute may have been removed. For template nodes, we look up the associated gap sequence and unfold each gap recursively. The last three rules in the figure define how a gap can be unfolded relative to a template node: either by following a template edge, by making an explicit template gap, or by removing the gap.

We define the language of a summary graph as

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

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

Compared with the definition of summary graphs in [8], a node now corresponds to at most one chardata sequence, element, or attribute—corresponding to the possible targets of XPath evaluation. Furthermore, we have added the $\mathit{removed}$ component of the gap presence map to model the `close` operation. Since every summary graph expressed according to the old definition can be transformed into one that fits into the new definition by splitting templates into individual nodes, we say that the latter one defines *normalized* summary graphs.

IV. Modeling XML Operations on Summary Graphs

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

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

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

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

`constant`: We show below in Section IV-A how to construct a summary graph SG_{xml} for a given template constant `[[`xml`]]`.

`plug`: All four variants of `plug` operations are modeled essentially as in [8], and the details are deferred to Appendix III. Intuitively, a template plug invocation exp_1`.plug(`g, exp_2`)` is modeled by adding template edges from nodes with open g gaps in $\widehat{\Delta}(exp_1)$ to roots in $\widehat{\Delta}(exp_2)$. A string plug is modeled by collecting the possible strings into the associated chardata node.

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

`select` and `gapify`: The modeling of these operations is based on a technique for symbolic XPath evaluation on summary graphs described in Section IV-C.

`group`: An array of XML templates is modeled by a single summary graph that approximates the array entries. To model an instance of the `group` operation, let n denote its template node and define $gaps(n) = g_1 g_2$ where g_1 and g_2 are fresh unique gap names. If $\widehat{\Delta}(exp) = (R, T, S, P)$ then we define $\widehat{\Delta}(\texttt{group(}exp\texttt{,}p\texttt{)}) = (\{n\}, T', S, P')$, where T' and P' are copies of T and P, respectively, with the following modifications: we add $(n, g_1, m) \in T'$ for each $m \in R$, $(n, g_2, n) \in T'$, and $n \in removed(P(g_i))$ for $i = 1, 2$. Intuitively, this models the output of a `group` operation as all possible concatenations of the input templates.

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

All transfer functions can be shown to be monotone.

Once the summary graphs are constructed, the `analyze` invocations are checked using a variation of the validation algorithm from [8], which validates the summary graph for the XML expression relative to the DTD. The original algorithm works on non-normalized summary graphs and DSD2 schemas, but it is easily adjusted to the present simpler setting. This is a conservative analysis of the summary graph: if it returns "valid", then it is guaranteed that all XML templates at that point are valid at runtime; otherwise, a useful error message is provided.

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

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

A. Summary Graphs for XML Template Constants

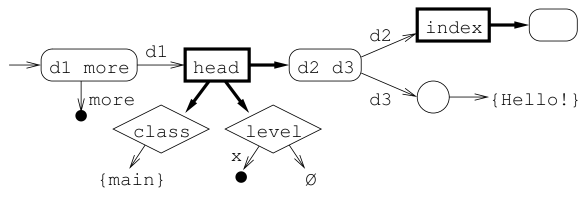
For the `constant` operations, we are given a template constant xml, and we need to construct a summary graph SG_{xml} such that $unfold(SG_{xml}) = \{xml\}$. This is trivial for the non-normalized summary graphs in [8] where each template constant corresponds to an individual summary graph node. For normalized summary graphs, the desired summary graph $SG_{xml} = (R, T, S, P)$ is the least one that satisfies the constraints generated from the following rules:

- For each element `<`$e\ldots$`>`$d_1 \ldots d_k$`</`e`>` in the template, let n denote the template node of the contents $d_1 \ldots d_k$ and define $gaps(n) = g_1 \ldots g_k$ where $g_i = h_i$ if $d_i =$ `<[`h_i`]>` and otherwise g_i is a fresh unique gap name. For each i, add $(n, g_i, m_i) \in T$ where m_i is the element node or chardata node of d_i.
- For the toplevel template contents corresponding to the template node r, we define $gaps(r)$ and add template edges in the same way as for element contents, and we define $R = \{r\}$.
- For every attribute a`="`s`"` corresponding to an attribute node n, add $S(n) = \{s\}$, and similarly for chardata.
- For every attribute gap a`=[`g`]` corresponding to an attribute node n, add $n \in open(P(g))$ and $agaps(P(g)) = \{\text{OPEN}\}$.
- For every template gap `<[`g`]>` belonging to a template node n, add $n \in open(P(g))$, $tgaps(P(g)) = \{\text{OPEN}\}$.
- Unless defined otherwise above, $agaps(P(g))$ and $tgaps(P(g))$ are set to $\{\text{CLOSED}\}$.

As an example, the template constant

```
<head class="main" level=[x]><index/>Hello!</head><[more]>
```

contains all possible template constructs. It is converted to the following summary graph:



Again, boxes represent element nodes, rounded boxes are template nodes, the circle is a chardata node, and the dots represent potentially open gaps. The diamonds are attribute nodes, and `d1`, `d2`, and `d3` are fresh gap names.

B. Converting DTD Schemas to Summary Graphs

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

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

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

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

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

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

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

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

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

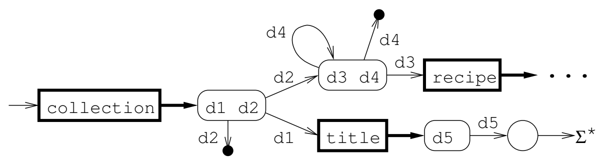
$(C)?$: For optional contents, let $(n, g, m) \in T$ for the template node m of C and add $n \in removed(P(g))$.

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

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

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

As an example, the DTD schema for recipe collections from Section II is converted to the following summary graph (abbreviated with "…"):



The gap names `d1`, …, `d5` are fresh names.

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

C. Symbolic XPath Evaluation

To model the XML operations that involve XPath, we symbolically evaluate a given XPath location path p on a summary graph $SG = (R, T, S, P)$. This evaluation is expressed by a function $eval$ that maps (SG, p) into a status map of the form $N_{\mathcal{E}} \cup N_{\mathcal{A}} \cup N_{\mathcal{C}} \rightarrow \mathcal{S}$ where $\mathcal{S} = \{\text{ALL}, \text{SOME}, \text{DEFINITE}, \text{NONE}, \text{NEVER}, \text{DONTKNOW}\}$. For a concrete unfolding $x \in \mathcal{L}(SG)$, a given element, attribute, or chardata node n from SG may correspond to a number of XML tree nodes in x. A concrete evaluation of p on x may select only some of those nodes. Informally, the possible values of $eval(SG, p)(n)$ have the following meaning:

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

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

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

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

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, $\sqsubseteq$, with DONTKNOW as top element, ALL and SOME above DEFINITE, and ALL and NONE above NEVER:

$$\begin{tikzcd}
 & & \text{DONTKNOW} \arrow[dll, dash] \arrow[d, dash] \arrow[drr, dash] & & \\
\text{NONE} & & \text{ALL} & & \text{SOME} \\
 & \text{NEVER} \arrow[ul, dash] \arrow[ur, dash] & & \text{DEFINITE} \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

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

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

$$eval(SG, p) = \left(path_p^{SG}(\sigma_0^{SG})\right)[root \mapsto \text{NONE}]$$

$$\sigma_0^{SG}(n) = \begin{cases} \text{DEFINITE} & \text{if } n = root \\ \text{NONE} & \text{if } root \rightsquigarrow n \\ \text{NEVER} & \text{otherwise} \end{cases}$$

The notation $f[x \mapsto y]$ denotes the function that is equal to f except that it maps x to y. The *reachability relation*, $\rightsquigarrow$, is defined as the transitive closure of the following rules:

$n \rightsquigarrow \mathit{contents}(n)$, $n \rightsquigarrow a$ for all $a \in \mathit{attr}(n)$, and $n \rightsquigarrow m$ for all $(n, g, m) \in T$.

A location path $p = s_1/\ldots/s_k$ is evaluated compositionally on each step:

$$\mathit{path}^{SG}_{s_1/\ldots/s_k} = \mathit{step}^{SG}_{s_k} \circ \cdots \circ \mathit{step}^{SG}_{s_1}$$

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

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

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

The function $\mathit{move}^{SG}_{\mathit{axis}}$ models the evalution of an axis:

$$\mathit{move}^{SG}_{\mathit{axis}}(\sigma)(n) = \begin{cases} \mathsf{ALL} & \text{if } \Psi^{SG}_{\mathit{axis}}(\sigma, n) \ \vee\ \sigma(n) = \mathsf{NEVER} \\ \mathsf{SOME} & \text{if } \exists m : m \overset{!}{\triangleright}_{\mathit{axis}} n \wedge \\ & \qquad \sigma(m) \sqsubseteq \mathsf{SOME} \\ \mathsf{DEFINITE} & \text{if the conditions for } \mathsf{ALL} \text{ and} \\ & \quad \mathsf{SOME} \text{ are both satisfied} \\ \mathsf{NONE} & \text{if } \forall m : m \overset{?}{\triangleright}_{\mathit{axis}} n \Rightarrow \\ & \qquad \sigma(m) \sqsubseteq \mathsf{NONE} \\ \mathsf{NEVER} & \text{if the conditions for } \mathsf{ALL} \text{ and} \\ & \quad \mathsf{NONE} \text{ are both satisfied} \\ \mathsf{DONTKNOW} & \text{otherwise} \end{cases}$$

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

$$\Psi^{SG}_{\mathit{axis}}(\sigma, n) = \begin{cases} \forall m : m \overset{?}{\triangleright}_{\mathit{axis}} n \Rightarrow \sigma(m) \sqsubseteq \mathsf{ALL} \\ \wedge\ n \neq \mathit{root} \quad \text{if } \mathit{axis} \in \{\texttt{child}, \texttt{attribute}\} \\ \psi^{SG}_{\sigma}(n) \quad \text{if } \mathit{axis} = \texttt{descendant-or-self} \end{cases}$$

where ψ^{SG}_{σ} is the least solution to the equation

$$\psi^{SG}_{\sigma}(n) = \\ \sigma(n) \sqsubseteq \mathsf{ALL} \ \vee\ \big(n \neq \mathit{root} \wedge \forall m : m \overset{?}{\triangleright}_{\texttt{child}} n \Rightarrow \psi^{SG}_{\sigma}(m)\big)$$

The function $\mathit{filter}^{SG}_{\mathit{test}}$ changes the status of a node n to NONE if the kind and name of n does not match test, unless the status is already NEVER in which case it is unchanged.

If pred is a location path p', then we define two families of status maps, σ'_n and σ''_n for each $n \in N$, by recursively applying path:

$$\sigma'_n(m) = \begin{cases} \sigma(n) & \text{if } m = n \\ \mathsf{NEVER} & \text{if } \sigma(m) = \mathsf{NEVER} \\ \mathsf{NONE} & \text{otherwise} \end{cases}$$

$$\sigma''_n = \mathit{path}^{SG}_{p'}(\sigma'_n)$$

From these status maps, we now define the function $\mathit{filter}^{SG}_{\mathit{pred}}$, which models the predicate filtering:

$$\mathit{filter}^{SG}_{p'}(\sigma)(n) = \begin{cases} \mathsf{NEVER} & \text{if } \sigma(n) = \mathsf{NEVER} \\ \mathsf{NONE} & \text{if } \sigma(n) \neq \mathsf{NEVER} \wedge \\ & \quad \forall m : \sigma''_n(m) \sqsubseteq \mathsf{NONE} \\ \sigma(n) & \text{if } \exists m : \sigma''_n(m) \sqsubseteq \mathsf{SOME} \\ \mathsf{DONTKNOW} & \text{otherwise} \end{cases}$$

This definition can be extended to also precisely model negated predicates and unions of node sets. If pred is not a location path, then $\mathit{filter}^{SG}_{\mathit{pred}}$ changes the status of a node n to $\mathsf{DONTKNOW}$ unless its status is already NONE or NEVER.

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

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

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

$$\mathit{HITS} = \{n \mid \mathit{eval}(SG, p)(n) \not\sqsubseteq \mathsf{NONE}\}$$

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

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

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

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

where, informally, $\mathit{LIVE} \subseteq N$ contains a node n if for every unfolding of SG all instances of n are certain to be retained by the operation; and similarly, DEAD contains the nodes that are certain to be removed. These sets can be computed by simple reachability analyses based on the status map $\mathit{eval}(SG, p)$.

The modeling of `gapify` is defined similarly:

$$\begin{aligned} &\widehat{\Delta}(\mathit{exp}\texttt{.gapify(}p, g\texttt{)}) = \\ &\quad \big(R, \\ &\quad T \setminus \{(n, h, m) \in T \mid m \in \mathit{ALL}\} \\ &\qquad \cup \{(n, h, t) \mid (n, h, m) \in T \wedge m \in \mathit{HITS}\}, \\ &\quad S[n \mapsto \emptyset \text{ for each } n \in \mathit{ALL} \cap (N_{\mathcal{C}} \cup N_{\mathcal{A}})], \end{aligned}$$

Example	Lines	Input	Output	SGs	SG Nodes	SG Edges	Max Space	Total Time	SG Space	SG Time
`ToUpper`	26	25	25	13	612	2,045	63 MB	10.8 s	5 MB	1.1 s
`Sorting`	43	25	25	13	606	2,686	61 MB	8.6 s	6 MB	1.2 s
`AddrBook1`	32	4	3	31	50	378	56 MB	9.0 s	2 MB	0.2 s
`AddrBook2`	17	5	4	14	49	215	55 MB	7.6 s	2 MB	0.2 s
`BankServlet`	88	5	1,201	23	48	1,008	74 MB	8.9 s	2 MB	0.4 s
`Country`	72	6	1,201	26	73	1,203	74 MB	9.2 s	2 MB	0.5 s
`Recipes`	137	25	1,201	100	748	10,987	81 MB	13.9 s	7 MB	2.6 s
`Article`	132	8	1,235	61	114	3,491	77 MB	9.7 s	3 MB	0.7 s
`BCedit`	190	9	9	46	169	1,945	97 MB	14.2 s	5 MB	0.6 s
`Tree`	73	15	24	82	183	4,921	61 MB	8.5 s	4 MB	1.0 s
`HTML2latex`	159	1,201	0	59	2,164	26,910	52 MB	11.9 s	14 MB	3.3 s
`CourseAdmin`	3,156	195	1,666	1,044	2,615	161,881	228 MB	74.3 s	45 MB	31.2 s

Fig. 3. Experimental results.

$$\begin{aligned}P'\big[g \mapsto (&open(P(g)) \cup \{t\} \cup (HITS \cap N_\mathcal{A}),\\ &removed(P(g)),\\ &merge(ANY_{N_\mathcal{E}\cup N_\mathcal{C}}, tgaps(P(g))),\\ &merge(ANY_{N_\mathcal{A}}, agaps(P(g))))\big]\big)\end{aligned}$$

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

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

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

and the function $merge$ is the same as in [8]:

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

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

It is possible to increase precision for the modeling of `gapify` by also considering the property of the semantics of this operation that an XML tree node is never considered selected if an ancestor is. We model this property by inserting an application of a function $sharpen$ to the result of each application of $eval(SG,p)$. Intuitively, $sharpen$ traverses SG from the roots and, for instance, converts ALL to NONE for a node n if it is able to determine that n has an ancestor of status ALL in every possible unfolding. We omit the formal definition.

V. Implementation and Experiments

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

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

The `ToUpper` benchmark changes all XML recipe titles to upper case using the DTD from Section II. The `Sorting` benchmark is the application that sorts a recipe collection in lexicographic order of the titles. The `AddrBook1` benchmark is the standard XDuce example, and the `AddrBook2` benchmark is a variation with a more realistic XML design. The `BankServlet` is a Servlet that produces an XHTML account summary from an XML database. The `Country` benchmark implements an XSLT 2.0 use case in which a collection of cities is grouped according to their country. The `Recipes` benchmark emulates an XSLT stylesheet producing XHTML from XML recipes; however, our version statically guarantees that the output is valid XHTML. The `Article` benchmark manipulates articles represented in XML. The `BCedit` benchmark from [18] is originally based on JDOM and implements a graphical editor on XML business cards. The `Tree` benchmark implements all queries in the corresponding XQuery use case [19]. Both `ToUpper`, `Country`, and `Tree` are shown in Appendix II. The `HTML2latex` benchmark is borrowed from the ℂDuce project [20]. Finally, the `CourseAdmin` benchmark is a real application implementing a generic course administration Web service, using specialized XML languages for representing data about schedules, students, teachers, and homeworks.

Figure 3 shows experimental results. “Lines” is the the number of lines in a desugared self-contained application, “Input” is the total number of lines of the DTD schemas involved in `cast` and `get` operations, “Output” is the the total number of lines of the DTD schemas involved in `analyze` operations, “SGs” is the total number of summary graphs computed during analysis, “SG Nodes” is the total number of summary graphs nodes allocated, “SG Edges” is the number of summary graph edges allocated. The maximal memory and overall time consumption are shown in the “Max Space” and “Total Time” columns, and the memory and time consumption during summary graph construction and analysis are shown in the “SG Space” and “SG Time” columns. The analysis time is measured in seconds, and the memory consumption in megabytes. Most of the large difference between “SG Space” and “Max Space” is consumed by the Soot framework [21], [22], which we use to construct control flow graphs from class files. Similarly, Soot has a startup time of around 7 seconds, which is the main cause of the differences between “SG Time” and “Total Time”. A possible remedy is to build a more specialized class analysis tool on top of another framework such as Recoder [23].

All experiments are performed on a 2.4 GHz Pentium IV with 1 GB RAM running Linux and J2SE version 1.4.2. The source for all benchmarks is available from `http://www.brics.dk/Xact/`.

Most of these benchmarks are small but demonstrate complex XML transformations that are typically expressed in specialized languages. As indicated by the `CourseAdmin` application, the number of lines of code is only a weak measure of the complexity of the analysis task. The sizes of the involved DTDs and the computed summary graphs more truly reflects the “XML complexity” of an application. Large applications will typically involve a limited number of XML transformation, each of which will be reminiscent of the benchmarks given above. A real strength of our analysis technique is the ability to extract the essential information from large Java programs and focus the analysis on such subtasks.

The precision of our analysis is reflected in the number of false errors flagged during analysis, which in all cases turns out to be zero. Furthermore, during the programming of the examples, the analysis found several actual errors that were subsequently corrected.

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

VI. Conclusion

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

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

We plan to integrate XACT into frameworks for making Web services, in particular JWIG and Servlets, and to make the system available as a stand-alone package for XML transformation in Java. Our prototype implementation is available online at `http://www.brics.dk/Xact/`.

Appendix I
Survey of Related Work

There exists a wide range of approaches for defining XML transformations, originating from database, hypertext, and programming language communities. These approaches are in the following divided into techniques for *general-purpose* programming languages and for tailor-made *domain-specific* languages. A general introduction to the XML type checking problem is given in [24].

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

Techniques for general-purpose languages

The approaches of representing XML data as strings or DOM trees, as mentioned in the introduction, fit into the category of techniques for general-purpose languages. Building XML documents by concatenating string fragments is commonly used in the presentation layer of interactive Web services, for example with Servlets [25]. This primitive approach does not assist the programmer in avoiding mismatching tags or improper escaping of special characters, and it does not support deconstruction of documents.

Presently, there are XML libraries with parsers and DOM-like functionality for all major (and also many less widely used) programming languages. Examples for Java include JDOM [7], TrAX [26], and JAXP [27]. Such libraries view XML data as tree structures and provide operations for local traversal and manipulation. This is a powerful approach that permits the full underlying programming language to be involved in the XML processing. Wellformedness of the involved XML data comes for free when working on the tree level. However, it is still a low-level approach for a number of reasons: 1) traversing or modifying a DOM tree is expressed via primitive operations, for example taking a single step in the tree from an element to its first child element. More complex operations therefore tend to require relatively much code, compared to e.g. XSLT, which is described below; 2) there is no tool support for analyzing the programs at compile-time to verify that transformation output is guaranteed to be valid at runtime or that the transformations succeed without runtime errors. XML is regarded as one homogeneous type without considering schemas. The processing is completely independent from the schema information, so, for example, a schema may contain the information that A elements cannot occur as children of B elements, but failed attempts to select an A child element of a B element in a program will not be detected until runtime.

SAX [28] is event-based rather than tree-based. This approach is suitable for streaming processing of large documents, but static validity is not considered.

To attack the problem of statically guaranteeing validity of the transformation output, a number of systems attempt to model XML transformation using pre-existing type systems in general-purpose programming languages. Examples based on functional languages are HaXml [29] and WASH/CGI [30], both embedding DTD into Haskell. In contrast to HaXml, WASH/CGI does not support deconstruction of XML values. In return, WASH/CGI allows the use of generic combinators, which the type-safe approach in HaXml does not.

With this approach, type checking of XML transformations comes for free via the type system in the host language. However, these type systems are usually not strong enough to capture all requirements specified in a schema without sacrificing soundness, performance, or flexibility [31], even with a simple schema language as DTD. Another problem is that type errors are reported at the level of the underlying host language, which can make them difficult to understand for the programmer.

Other systems are targeted at object-oriented languages, typically Java. Castor [32] and the more recent JAXB [33] are XML data binding frameworks for Java. From a schema written in certain subsets of XML Schema they can generate a collection of Java classes representing an object model of the corresponding XML documents. XML data may then be processed as Java objects at a higher abstraction level than e.g. JDOM. Methods for marshalling and unmarshalling are automatically generated, and the mapping between XML and Java can be controlled by specifying explicit bindings. Relaxer [34] is a similar tool but for the RELAX schema language. For all three systems, there is no static guarantee that a constructed document will satisfy all the requirements of the given schema.

The SNAQue tool [35] provides a variant of data binding that does not take schemas into account. From an XML document and a programming language type, it extracts a program value. Projector [36] is a related extension of JavaScript mixing typed and untyped programming.

The approach described in [37] contains a data binding system for languages with powerful types with streams, tuples, and unions, which allow schemas to be encoded with high precision. A type checking algorithm is currently being implemented but is yet unpublished. Many other data binding tools are described in [38].

Domain-specific languages

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

The predominant DSL for XML transformation is XSLT [39], a declarative language based on pattern matching and template instantiation. Although designed primarily for hypertext stylesheet applications, it is more widely applicable, for example, for simple database operations. XSLT uses XPath for pointing and pattern matching. Schemas for the input and output languages are ignored by XSLT 1.0 processors, so no type checking is performed. XSLT 2.0 [40] is currently being designed. It uses types from XML Schema but only supports dynamic validation. (*"It is implementation-defined whether type errors are signaled statically."* [40]) XSLT stylesheets can to a large extent easily be converted into XACT programs by turning XSLT templates into methods that return XACT templates. The XSLT pattern matching feature, which determines the templates to instantiate, does not have a direct counterpart in XACT where the control-flow is more explicit.

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

XQuery [41] can be viewed as a generalization of SQL to the richer data model of XML. It is a functional language with optional types using a considerable subset of XML Schema as basis for its type system [42], which supports static type inference and checking. Although still at working draft level with many open issues, XQuery is an ambitious project and receives much attention.

XDuce [31] is a simplistic functional language based on regular expression types, which are a natural generalization of DTD schemas, and a corresponding mechanism for pattern matching. It supports a local form of type inference where types are specified explicitly for function arguments but inferred for pattern matching. In its current version, XDuce does not have higher-order functions or parametric polymorphism, and the type system does not model element attributes or unordered data. ℂDuce [20] extends XDuce into a full programming language and adds higher-order functions and other language features. The ideas from XDuce, which have also influenced the design of the XQuery type system, are currently being integrated into C# in the Xtatic project with similar goals as ours [43]. Another related language is Circus-DTE [44], which is a simple transformation language with pattern matching and type-checking mechanisms reminiscent of those in XDuce.

XMλ [45] is a functional language related to HaXml and WASH/CGI. Its type system uses a notion of type-indexed rows to model DTD. Whereas subtyping is an essential aspect in XDuce, XMλ is based on parametric polymorphism. Apparently, no implementation of XMλ is available.

The language *fxt* [14] is closely related to XSLT but uses a strictly top-down processing model and a clean pattern matching mechanism that corresponds to regular languages. Another attempt to redesign XSLT is SXSLT [46], based on Scheme. Both *fxt* and SXSLT focus on language design and, as XSLT, do not provide type checking.

The type checking problem has been studied at a more theoretical level for *k-pebble tree transducers* [47], a framework for modeling decidable tree transformations in, for example, fragments of XQuery and XSLT. A less expressive formalism for top-down transformations is investigated in [48], and another related approach is proposed in [49] for type checking a subset of XSLT using tree automata. In [50], a simple XML transformation system based on macro expansion is described, and it is shown that exact type checking with DTD is decidable for this system. The query language *loto-ql* permits inference of output schemas from input schemas using a generalization of DTD to context-free languages [51].

Finally, we mention the recent XOBE language [52], which is closely related to our approach. XOBE is also an extension of Java, it has a notion of XML templates resembling that of JWIG and XACT, and it too uses XPath to select parts of XML trees. XOBE uses a type system based on regular hedge grammars, whereas we rely on dataflow analysis using summary graphs to obtain static guarantees. However, there are a number of more essential differences: XOBE requires all XML variables to be explicitly typed with element names, unlike our approach. Lists of mixed elements can be described by unordered content models, not by general regular expressions. XML trees in XOBE can only be constructed bottom-up. In contrast, the template mechanism in JWIG and XACT is higher-order in the sense that templates can contain named gaps that can be filled in any order, possibly with templates containing other gaps. Finally, our *gapify* construct has no counterpart in XOBE. These issues make XACT more flexible in practice.

APPENDIX II
SYNTACTIC SUGAR FOR XACT

The XACT language permits some syntactic sugar on top of the basic operations. First, we allow special syntax for template constants, which may be written in `[[...]]` without the otherwise mandatory escape characters. Similarly, arguments of types `Gap` and `XPath` may be written directly without explicit calls to constructors, and `DTD` references can be written as strings. Additionally, we allow some simple abbreviations for common operations:

```
smash(xs) ≡ xs.length>0 ? group(xs,.[false()])[0] : [[]]
x.roots() ≡ x.select(*)
x.text() ≡ smash(x.select(text())).toString()
x.attribute(a) ≡ smash(x.select(@a)).toString()
x.has(p) ≡ x.select(p).length>0
x.size() ≡ x.roots().length
x.delete(p) ≡ x.gapify(p,g)
x.apply(p,f) ≡ x.gapify(p,g).plug(g,[]f(x.select(p)))
```

The `smash` operation concatenates an array of templates into a single template; `roots` builds an array with one entry for each root element in the given template; `text` extracts the top-level character data of a template; `attribute` extracts the value of an attribute; `has` checks whether specific nodes are present; `size` counts the number of root elements in a template; and `delete` effectively removes the specified nodes from a template. The `apply` operation applies a transformation to the specified nodes, under the assumption that these nodes have disjoint subtrees. If f is a local method accepting exactly one argument of type `XML` and whose result is also of type `XML`, then `[]`f abbreviates a new local method that accepts and returns arguments of type `XML[]` and applies f to each array entry. A recursive variant of `apply` works without the disjointness restriction.

Finally, a *code gap* is syntactic sugar for a gap and a plug operation: `<{`c`}>`, where c is an expression of type `String` or `XML`, abbreviates a gap `<[g]>` and a plug operation where the value of c is plugged into g. Alternatively, c can be a statement returning a value of type `String` or `XML`. Code gaps can also occur as attributes using the notation *name*=`{`c`}`.

Consider a method `upperTitle` that creates a copy of a recipe collection in which all titles are raised to upper case. We use the DTD schema from Section II to model recipes. The following sugared syntax

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

then abbreviates the more cumbersome basic syntax:

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

These syntactic extension to Java can be implemented using the Metafront tool [53].

The following complete example implements the recursive TREE Q6 query from the XQuery use cases [19]:

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

```
            <{summary(x.select(book/section))}>
          </toc>]]
        .analyze("Q6.dtd").toString();
}
```

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

The next example shows how a group-like transformation task inspired by use cases in the XSLT 2.0 requirement specification [54] can be solved with XACT. The task is to produce an XHTML document where cities are grouped in a table according to their country, and with the total population computed for each group. The format for cities is the one exemplified in Section II. Since XHTML documents have the same basic structure it is beneficial to provide the following template:

```
XML xhtml = [[
  <html>
   <head><title><[title]></title></head>
   <body><[body]></body>
  </html>
]];
```

The transformation task is accomplished by

```
XML getRows(XML[] xs) {
  XML[] rs = new XML[xs.length];
  for (int i=0; i<xs.length; i++) {
    XML[] cities = xs[i].select("city");
    String country="", names="";
    int pop=0;
    for (int j=0; j<cities.length; j++) {
      country=cities[j].attribute("country");
      names+=cities[j].attribute("name")+" ";
      pop+=Integer.parseInt(cities[j].attribute("pop"));
    }
    rs[i] = [[<tr>
               <td><{country}></td>
               <td><{names}></td>
               <td><{pop}></td>
              </tr>]];
  }
  return XML.smash(rs);
}
XML getXHTML(String s) {
  XML table = xhtml.plug(head,"Groups of Cities");
                   .plug(body,[[<table><[rows]></table>]]);
  XML x = XML.get(s,"cities.dtd");
  XML[] cs = x.select("/cities/city");
  XML[] gs = XML.group(cs,"city/@country");
  return table.plug(rows,getRows(gs));
               .analyze("xhtml1-transitional.dtd");
}
```

Note how the result is constructed in a top-down fashion using the `plug` operation. This programming style is appropriate when sub-templates of the transformation, such as `xhtml` and `table` in the above example, are candidates for reuse within the transformation.

APPENDIX III
MODELING AND CHECKING PLUG OPERATIONS

This appendix shows the transfer function for `plug` operations and the compile-time test for absence of runtime errors at these operations.

A template plug invocation, $x\,.\,\texttt{plug}(g, y)$ where y has type `XML`, is modeled by adding template edges from nodes with open g gaps in $\widehat{\Delta}(x)$ to roots in $\widehat{\Delta}(y)$. A string plug, that is, where y has type `String`, is modeled by collecting the possible strings of y at the program point ℓ into the associated chardata node:

$$\widehat{\Delta}(x\,.\,\texttt{plug}(g,y)) = \begin{cases} \mathit{tplug}(\widehat{\Delta}(x), g, \widehat{\Delta}(y)) & \text{if } y \text{ has type } \texttt{XML} \\ \mathit{splug}(\widehat{\Delta}(x), g, \mathit{string}_\ell(y)) & \text{if } y \text{ has type } \texttt{String} \end{cases}$$

We use the auxiliary functions *tplug*, *splug*, and string_ℓ:

$$\begin{aligned} &\mathit{tplug}((R_1,T_1,S_1,P_1), g, (R_2,T_2,S_2,P_2)) = \\ &\quad (R_1, \\ &\quad T_1 \cup T_2 \cup \{(n,g,m) \mid n \in \mathit{open}(P_1(g)) \;\wedge\; m \in R_2\}, \\ &\quad \lambda m.S_1(m) \cup S_2(m), \\ &\quad \lambda h.\text{if } h = g \\ &\qquad \text{then } (o_2, r_1 \cup r_2, t_2, a_2) \\ &\qquad \text{else } (o_1 \cup o_2, r_1 \cup r_2, \mathit{merge}(t_1,t_2), \mathit{merge}(a_1,a_2))) \end{aligned}$$

where $P_1(h) = (o_1, r_1, t_1, a_1)$, $P_2(h) = (o_2, r_2, t_2, a_2)$, *merge* is as defined in Section IV-C, and:

$$\begin{aligned} &\mathit{splug}((R,T,S,P), g, L) = \\ &\quad (R, \\ &\quad T \cup \{(n,g,c) \mid n \in \mathit{open}(P(g)) \cap N_{\mathcal{T}}\}, \\ &\quad S[n \mapsto S(n) \cup L \text{ for } n \in (\mathit{open}(P(g)) \cap N_{\mathcal{A}}) \cup \{c\}] \\ &\quad P[g \mapsto (\emptyset, \mathit{removed}(P(g)), \{\mathsf{CLOSED}\}, \{\mathsf{CLOSED}\})]) \end{aligned}$$

where c is the chardata node corresponding to the occurrence of the `plug` operation.

A separate program analysis, see [17], provides a regular string language over the Unicode alphabet for each occurrence of a string expression in the program. The set $\mathit{string}_\ell(y)$ thus contains an upper approximation of the set of strings that the expression y may evaluate to at the program point ℓ at runtime.

The *tplug* function models `plug` operations where the second operand is an XML template expression. It finds the summary graphs for the two sub-expressions and combines them as follows: The roots are those of the first graph since it represents the outermost template. The template edges become the union of those in the two graphs plus a new edge from each node that may have open gaps of the given name to each root in the second graph. The string edge sets are simply joined without adding new information. For the gaps that are plugged into, we take the gap presence information from the second graph, except for the *removed* component, which is joined from the two summary graphs. For the other gaps we use the *merge* function to mark gaps as “definitely open” if they are so in one of the graphs and otherwise take the least upper bound.

The *splug* function models `plug` operations where the second operand is a string expression. It adds an edge from each template node with an open gap of the given name to the chardata node that corresponds to the operation. The string edge map is updated by adding the set of strings obtained by the string analysis for the string expression to the chardata node and to each attribute with an open gap of the given name. The gap presence map is updated to mark the gaps as “definitely closed”.

The array variants of `plug` are modeled as above, except that we need to model the case where the given array is shorter than the number of gaps of the given name and the remaining gaps are filled with the empty string. This is accomplished by adding the empty string to the string edge map for the chardata node corresponding to the operation and, for the template array variant, also adding a template edge from each template node with an open gap of the given name to the chardata node.

As mentioned in Section II, one of the compile-time guarantees that our analysis can provide is that `plug` XML templates are never plugged into attribute gaps. A safe approximation of this information can be extracted from the summary graphs: For a specific plug operation $x\texttt{.plug}(g, y)$ where y has type `XML` or `XML[]`, consider the summary graph (R, T, S, P) given by the data-flow analysis for the expression x. We now check the plug operation simply by inspecting that the following condition is satisfied:

$$agaps(P(g)) = \{\textsf{CLOSED}\}$$

If a violation is detected, a helpful error message can be generated. Additionally, if

$$agaps(P(g)) \cup tgaps(P(g)) = \{\textsf{CLOSED}\}$$

then the plug operation will never have any effect because the g gap is never present in the x template. In this case, a warning is generated.

References

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

[2] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "XML Schema part 1: Structures," May 2001, W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.

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

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

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

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

[7] J. Hunter and B. McLaughlin, "JDOM," 2001, `http://jdom.org/`.

[8] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Extending Java for high-level Web service construction," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 6, pp. 814–875, November 2003.

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

[10] J. Clark and S. DeRose, "XML path language," November 1999, W3C Recommendation. `http://www.w3.org/TR/xpath`.

[11] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Static analysis for dynamic XML," BRICS, Tech. Rep. RS-02-24, May 2002, Presented at Programming Language Technologies for XML, PLAN-X, October 2002.

[12] C. Kirkegaard, A. S. Christensen, and A. Møller, "A runtime system for XML transformations in Java," BRICS, Tech. Rep. RS-03-29, October 2003.

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

[14] A. Berlea and H. Seidl, "Transforming XML documents using fxt," *Computing and Information Technology, Special Issue on Domain-Specific Languages*, vol. 10, no. 1, pp. 19–35, 2002.

[15] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer-Verlag, October 1999.

[16] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica*, vol. 7, pp. 305–317, 1977, Springer-Verlag.

[17] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium, SAS '03*, ser. LNCS, vol. 2694. Springer-Verlag, June 2003, pp. 1–18.

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

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

[20] V. Benzaken, G. Castagna, and A. Frisch, "CDuce: a white paper," October 2002, Presented at Programming Language Technologies for XML, PLAN-X.

[21] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot – a Java optimization framework," in *Proc. IBM Centre for Advanced Studies Conference, CASCON '99*. IBM, November 1999.

[22] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for Java," in *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00*, October 2000.

[23] A. Ludwig *et al.*, "Recoder," 2002 March, `http://recoder.sourceforge.net/`.

[24] D. Suciu, "The XML typechecking problem," *ACM SIGMOD Record*, vol. 31, March 2002.

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

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

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

[28] D. Brownell, *SAX2*. O'Reilly & Associates, January 2002.

[29] M. Wallace and C. Runciman, "Haskell and XML: Generic combinators or type-based translation?" in *Proc. 5th ACM SIGPLAN International Conference on Functional Programming, ICFP '99*, September 1999.

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

[31] H. Hosoya and B. C. Pierce, "XDuce: A statically typed XML processing language," *ACM Transactions on Internet Technology*, vol. 3, no. 2, 2003.

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

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

[34] M. Fitzgerald, "Relaxer tutorial," `http://www.relaxer.org/doc/tutorial/tutorial.html`, 2003.

[35] F. Simeoni, P. Manghi, D. Lievens, R. H. Connor, and S. Neely, "An approach to high-level language bindings to XML," *Information & Software Technology*, vol. 44, no. 4, pp. 217–228, 2002, Elsevier.

[36] R. Connor, D. Lievens, F. Simeoni, S. Neely, and G. Russell, "Projector – a partially typed language for querying XML," October 2002, Presented at Programming Language Technologies for XML, PLAN-X.

[37] E. Meijer and W. Schulte, "Unifying tables, objects and documents," in *Proc. Declarative Programming in the Context of OO Languages, DP-COOL '03*, 2003.

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

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

[40] M. Kay, "XSL transformations (XSLT) version 2.0," May 2003, W3C Working Draft. `http://www.w3.org/TR/xslt20/`.

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

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

[43] V. Gapayev and B. C. Pierce, "Regular object types," in *Proc. 17th European Conference on Object-Oriented Programming, ECOOP'03*, ser. LNCS, vol. 2743. Springer-Verlag, July 2003.

[44] J.-Y. Vion-Dury, V. Lux, and E. Pietriga, "Experimenting with the Circus language for XML modeling and transformation," in *Proc. ACM Symposium on Document Engineering, DocEng '02*, November 2002.

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

[46] O. Kiselyov and S. Krishnamurthi, "SXSLT: Manipulation language for XML," in *Proc. 5th International Symposium on Practical Aspects of Declarative Languages, PADL '03*, January 2003.

[47] T. Milo, D. Suciu, and V. Vianu, "Typechecking for XML transformers," *Journal of Computer and System Sciences*, vol. 66, February 2002, Special Issue on PODS '00, Elsevier.

[48] W. Martens and F. Neven, "Typechecking top-down uniform unranked tree transducers," in *9th International Conference on Database Theory*, ser. LNCS, vol. 2572. Springer-Verlag, January 2003.

[49] A. Tozawa, "Towards static type checking for XSLT," in *Proc. ACM Symposium on Document Engineering, DocEng '01*, November 2001.

[50] T. Perst and H. Seidl, "A type-safe macro system for XML," in *Proc. Extreme Markup Languages*, August 2002.

[51] Y. Papakonstantinou and V. Vianu, "DTD inference for views of XML data," in *Proc. 19th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, PODS '00*, May 2000.

[52] M. Kempa and V. Linnemann, "Type checking in XOBE," in *Proc. Datenbanksysteme für Business, Technologie und Web, BTW '03*, ser. LNI, vol. 26, February 2003.

[53] C. Brabrand, M. I. Schwartzbach, and M. Vanggaard, "The metafront system: Extensible parsing and transformation," in *Proc. 3rd ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications, LDTA '03*, April 2003.

[54] S. Muench and M. Scardina, "XSLT requirements version 2.0," February 2001, W3C Working Draft. `http://www.w3.org/TR/xslt20req`.