Basic Research in Computer Science

BRICS RS-03-29 Christensen et al.: A Runtime System for XML Transformations in Java

A Runtime System for XML Transformations in Java

Aske Simon Christensen
Christian Kirkegaard
Anders Møller

BRICS Report Series **RS-03-29**

ISSN 0909-0878 **October 2003**

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

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/03/29/`

A Runtime System for XML Transformations in Java

Aske Simon Christensen, Christian Kirkegaard, and Anders Møller*
BRICS†, Department of Computer Science
University of Aarhus, Denmark
{aske,ck,amoeller}@brics.dk

Abstract

We show that it is possible to extend a general-purpose programming language with a convenient high-level data-type for manipulating XML documents while permitting (1) precise static analysis for guaranteeing validity of the constructed XML documents relative to the given DTD schemas, and (2) a runtime system where the operations can be performed efficiently. The system, named XACT, is based on a notion of immutable XML templates and uses XPath for deconstructing documents. A companion paper presents the program analysis; this paper focuses on the efficient runtime representation.

1 Introduction

There exists a variety of approaches for programming transformations of XML [6] documents. Some work in the context of a general-purpose programming language; for example, JDOM [14], which is a popular package for Java allowing XML documents to be manipulated using a tree representation. A benefit of this approach is that the full expressive power of the Java language is directly available for specifying the transformations. Another approach is to use domain-specific languages, such as XSLT [9], which is based on notions of templates and pattern matching. This approach often allows more concise programs that are easier to write and maintain, but it is difficult to combine it with more general computations, access to databases, communication with Web services, etc.

Our goal is to integrate XML into general-purpose programming languages to make development of XML transformations easier and safer to construct. We propose XACT, which integrates XML into Java through a high-level data-type representing immutable XML fragments, a runtime system that supports a number of primitive operations on

*Supported by the Carlsberg Foundation contract number ANS-1069/20.
†Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

such XML fragments, and a static analysis for detecting programming errors related to the XML operations.

The main contribution of this paper is the description of the XACT runtime system. We present a suitable runtime representation for XML templates that efficiently supports the operations in the XACT API. The companion paper [16] contains a description of the static analysis of XACT programs.

We first, in Section 2, describe the design of the XACT language and motivate our design choices. Section 3 then gives a brief overview of the results from [16] about providing static guarantees for XML transformations written in XACT. Section 4 presents our runtime system and discusses time and space complexity of the operations.

The most closely related work is that on XDuce and Xtatic by Hosoya, Pierce, and Gapayev [13, 11], XQuery by Fernández, Siméon, Wadler, and others [3], and WASH/CGI by Thiemann [21]. XDuce is a functional language for defining XML transformations based on regular expression types and a corresponding mechanism for pattern matching. The Xtatic project aims to integrate the ideas from XDuce with the object model of C#. XQuery is a functional language that can be viewed as a generalization of SQL to the richer data model of XML. WASH/CGI models XML transformations in Haskell by embedding DTD into Haskell's type system. The paper [16] contains a comprehensive survey of the relation between these and other projects and XACT.

2 The XACT Language

The XACT language introduces XML transformation facilities into the Java programming language such that XML documents, from a programmer's perspective, are first-class values on equal terms with basic values, such as booleans, integers, and strings. Programmers can thereby combine the flexibility and power of a general-purpose programming language with the ability to express XML manipulations at a high level of abstraction. This combination is convenient for many typical transformation tasks. Examples are transformations that rely on communication with databases and complex transformation tasks, which may involve advanced control-flow depending on the document structure. In these cases, one applies XACT operations while utilizing Java libraries, for example, the sorting facilities, string manipulations, HTTP communication, etc. We choose to build upon Java because it is widely used and a good representative for the capabilities of modern general-purpose programming languages.

We build XML documents from *templates* as known from the JWIG language [8]. This approach originates from MAWL [17, 1] and `<bigwig>` [5], and was later refined in JWIG, where it has shown to be a powerful formalism for XHTML document construction in Web services. Our aim is to extend the formalism to general XML transformations where both construction and deconstruction is supported.

A template is a well-formed XML fragment containing named gaps: *template gaps* occur in place of elements, and *attribute gaps* occur in place of attributes. The notation for templates is given by xml in the following grammar:

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

Here, str denotes a string of XML character data, $name$ denotes a qualified XML name, g denotes a gap name, and $value$ denotes an XML attribute value. Construction of a larger template from a smaller one is accomplished by *plugging* values into its gaps. The result is the template with all gaps of a given name replaced by values. This mechanism is flexible because complex templates can be built and reused many times. Gaps can be plugged in any order; construction is not restricted to be bottom-up, in contrast to traditional tree-like models, such as XDuce.

Deconstruction of XML data is also supported in XACT. An off-the-shelf language for addressing nodes within XML trees is available, namely W3C's XPath language [10]. XPath is widely used and has despite its simplicity shown to be versatile in existing technologies, such as XSLT and XQuery. The XACT deconstruction mechanism is also based on XPath. We have identified two basic deconstruction operations, which are powerful in combination with plugging. The first is *select*, which returns the subtemplates addressed by an XPath expression. The second is *gapify*, which replaces the subtemplates addressed by an XPath expression with gaps. Select is convenient because it permits us to pick subtemplates for further processing. Gapify permits us to dynamically introduce gaps, which is important for a task such as performing minor modifications in an XML tree. Altogether, this constitute an algebra over templates, which allows typical XML manipulations to be expressed at a high level of abstraction.

We have chosen a value-based programming model as in functional languages. This model is generally more "clean" since operations have no side-effects, and templates are thought of as unchangeable values. A Java class that implements the value-based model is said to be *immutable*. Such classes are favored because their instances are safe to share, value-factories can be implemented, and tread-safety is guaranteed [2]. All Java value classes, such as `Integer` and `String`, are for these reasons immutable. Our templates inherit the properties and benefit by being easier to use and less prone to error than mutable frameworks, such as JDOM and JAXP [20].

The Java class `XML`, which represents templates, has the methods shown in Table 1. The class is immutable, so the value represented by a given template object is never altered after instantiation. All parameters of type `Gap`, `XPath` and `DTD` are assumed to be constants and may be written as strings.

The static `constant` method creates an `XML` instance from a constant string argument, and `toString` returns the string representation of an `XML` instance. The syntax for templates is the one given by the grammar above. The `get` method constructs a template from a non-constant string, typically originating from some external data source, and checks the result for validity with respect to the given DTD schema. In addition,

`static XML constant(String` s`)`	- creates a template from the constant string s
`String toString()`	- returns the textual representation of this template
`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 x into all g gaps in this template
`XML plug(Gap` g`, String` s`)`	- as the previous operation, but for string
`XML plug(Gap` g`, XML[]` xs`)`	- inserts the entries in xs into the g gaps in this template
`XML plug(Gap` g`, String[]` ss`)`	- as the previous operation, but for string entries
`XML[] select(XPath` p`)`	- returns the array of subtemplates hit by p
`XML[] cut(XPath` p`)`	- as the previous, but returns only maximal disjoint subtemplates
`XML gapify(XPath` p`, Gap` g`)`	- replaces all subtemplates hit by p by g gaps
`XML close()`	- returns this template with all gaps removed
`String text()`	- returns the concatenation of top level chardata
`XML cast(DTD` d`)`	- throws runtime exception if this template is invalid relative to d
`XML analyze(DTD` d`)`	- instructs the analyzer to statically validate this template relative to d
`static XML smash(XML[]` xs`)`	- merges the entries of xs into a single template
`static XML get(String` s`, DTD` d`)`	- creates a template from the string s and checks validity relative to d

Table 1: Methods in the immutable `XML` class for performing basic XACT operations.

runtime validation of a template according to a given DTD schema is provided by the `cast` method, which serves the same purpose as the usual cast operations in Java. Both `get` and `cast` throw a runtime exception in case the given template is invalid.

The `equals` method determines equality of `XML` instances, and the `hashCode` method returns a consistent hash code for an `XML` instance.

Template construction is provided by the `plug` method, which is overloaded to accept a template, a string, or arrays of these as second parameter. Invoking the non-array variants will plug the given string or template into all occurrences of the given gap name. The array variants will, in document order, plug all occurrences of the given gap name with entries from the given array. If the array has superfluous entries these will be ignored, and conversely, the empty string will be plugged into superfluous gaps. An exception is thrown if one attempts to plug a template into an attribute gap.

Template deconstruction is provided by the `select`, `cut`, and `gapify` methods. Each method takes an XPath expression as parameter, which on evaluation returns a set of nodes within the given template. Invoking the `select` method gives an array containing all the subtemplates rooted at nodes in the XPath evaluation result. The `cut` method gives a similar array, but the entries are here required to be non-overlapping, such that if one node in the XPath evaluation result is an ancestor of another, then only the ancestor is considered. The returned subtemplates of `cut` are consequently maximal and disjoint. The `gapify` method returns a template where all subtemplates rooted at nodes in the XPath evaluation result have been replaced by gaps of the given name.

Extraction of character data from a template is provided by the `text` method, which returns the concatenation of top level character data as a string. The `close` method eliminates all gaps in a template, which is accomplished by removing template gaps and for attribute gaps, the whole attribute is removed. The result will by construction represent a well-formed XML document. Invoking the static `smash` method concatenates the entries of the given template array into a single template.

The `analyze` method instructs the compile-time analyzer to check for validity relative to a given DTD, as described in Section 3. This operation has no effects at runtime. A complete XML transformation typically begins with a number of `get` operations that read the transformation input and ends in `analyze` and `toString` operations that produce the transformation output and checks that it is valid.

In order to integrate XACT tightly with the Java language, we provide special syntax for template constants. This relieves programmers from tedious and error-prone character escaping. A template xml may be written `[[`xml`]]`, which after character escaping is equivalent to `XML.constant("`xml`")`. Transformations that use this syntax are desugared by a simple preprocessor, which is bundled with the XACT packages. Also, a number of useful macros for commonly occurring tasks are provided as methods of the `XML` class. For example, the `delete` macro effectively deletes the subtrees selected by an XPath expression by performing a `gapify` operation with a fresh gap name. The complete list of macros is presented in [16].

We now consider an example, originating from [12], where an address book is filtered in order to produce a phone list. An address book here consists of an `addrbook` root element, containing a sequence of `person` elements, each having a `name`, an `addr`, and an optional `tel` element as children. The filtration outputs a `phonelist` root element, containing a sequence of `person` elements, where only those having a `tel` child remains, and with all `addr` elements eliminated. The following method shows how this is implemented with XACT:

```
XML phonelist(XML book) {
  XML[] persons = book.select("/addrbook/person[tel]");
  XML list = XML.smash(persons).delete("//addr");
  return [[<phonelist><[list]></phonelist>]].plug("list",list);
}
```

One may additionally wish to sort the phone list alphabetically by name. Java has built-in sorting facilities for arrays, so this is accomplished by implementing a `Comparator` class, called `PersonComparator`, with the following `compare` method:

```
int compare(Object o1, Object o2) {
  XML x1 = (XML) o1, x2 = (XML) o2;
  String s1 = XML.smash(x1.select("/person/name/text()")).text();
  String s2 = XML.smash(x2.select("/person/name/text()")).text();
  return s1.compareTo(s2);
}
```

The phone list can then be sorted by inserting the following line into the `phonelist` method:

```
  Arrays.sort(persons, new PersonComparator());
```

The example shows how a complex transformation task can be easy and intuitive to express using the XACT language.

3 Static Guarantees

The design of XACT enables precise static analysis for guaranteeing absence of certain programming errors related to XML document manipulation. In the companion paper [16], we present a data-flow analysis that, at compile-time, checks the following correctness properties of an XACT program:

output validity — that each `analyze` operation is valid in the sense that the given XML template is guaranteed to be valid relative to the given DTD schema; and

plug consistency — that each `plug` operation is guaranteed to succeed, that is, templates are never plugged into attribute gaps.

Additionally, the analysis can detect and warn the programmer if the specified gap for a `plug` operation is never present and if an XPath expression in a `select`, `cut`, or `gapify` operation will never select any nodes.

The crucial property of XACT that makes this analysis feasible is that the XML templates are immutable. Analyzing programs that manipulate mutable data structures is known to be difficult [19, 18]. The absence of side-effects means that we do not have to model the complex aliasing relations that otherwise may arise.

Our analysis is an application of the standard data-flow analysis framework [15], but with a very specialized lattice structure consisting of *summary graphs*, originally introduced in [4] and later refined in [7] and [16]. Informally, a summary graph is a graph whose nodes represent elements, attributes, and gaps occurring in template constants or in DTD schemas, and whose edges represent template or string plug operations. A subset of the nodes are designated as roots. Additionally, a summary graph contains information about which template gaps and attribute gaps are present. Every summary graph represents a set of concrete XML templates: the language of a summary graph is the set of XML templates that can be obtained by unfolding the graph, starting from a root and plugging templates and strings into the gaps according to the edges and the gap presence information.

The notion of summary graphs constitutes a suitable abstraction of the concrete XML templates that appear at runtime. Each XACT operation can be modeled precisely as a transformation of summary graphs. For example, a template plug operation combines two summary graphs by adding appropriate edges; every template constant and DTD schema occurring in the given program can be converted into a corresponding summary graph; and XPath expressions can be modeled precisely by a process of symbolic evaluation on the summary graphs.

The analysis is conservative in the sense that it never misses an error, but it might report false errors. Our experiments in [16] indicate that the analysis is both precise and efficient enough to be practically useful, and that it produces helpful error messages if potential errors are detected.

4 Runtime System

We have now presented a high-level language for expressing XML transformations and briefly explained that the design permits precise static analysis. However, such a

framework would be of little practical value if the operations could not be performed efficiently at runtime. In this section, we present a data structure addressing this issue.

4.1 Requirements

To qualify as a suitable representation for XML templates in the XACT framework, our data structure must support the following operations:

- *Creation*: Given the textual representation of an XML template, build the structure representing the template.
- *Combination*: The `smash`, `plug` and `close` operations operate directly on XML templates and must be supported directly by the data structure.
- *Navigation*: The tasks of converting a template to its textual representation, checking the template for validity according to a given schema, or evaluating an XPath expression on a template, all require means for traversing the XML data in various ways. In general, we must have a mechanism for pointing at a specific node in the XML tree. We call such an XML pointer a *navigator*. It must support operations for moving this pointer around the tree. To support all XPath axis evaluations, we must be able to move to the *first child* and *first attribute* of an element node, the *parent* and *next/previous sibling* of any tree node, and the *next/previous attribute* of an attribute node.
- *Extraction*: The result of evaluating an XPath expression on the structure, using its navigation mechanism, is a set of navigators. From this set of navigators, we must be able to obtain the result of the `select`, `cut` and `gapify` operations.

A naive data structure that trivially supports all of these operations is an explicit XML tree with *next*, *previous*, *parent* and *firstchild* pointers in all nodes, similarly to a JDOM tree. If such a data structure is used, we are forced to copy all parts of the operand structures that constitute parts of the result in order to adhere to the immutability constraint. The doubly-linked nature of the structure prohibits any sharing between individual XML values. The running times for the XACT operations operating on such a structure would thus be at least linear in the size of the result, which is certainly unsatisfactory.

4.2 The basic approach

The main problem with the doubly-linked tree structure is that it prevents sharing between templates. To enable sharing, we use a singly-linked tree, that is, a tree with only *firstchild* and *next* pointers but without the *parent* and *previous* pointers. This structure permits sharing as follows: Whenever a subtree of an operand occurs as a subtree of the result, the corresponding pointer in the result simply points to the original operand subtree and thus avoids copying that subtree.

The `smash` operation is trivial in this representation. We simply point to the roots of all operands. This takes time proportional to the number of templates.

To perform a non-array `plug` operation, x`.plug(`g`,` y`)`, we copy just the portion of x that is not part of a subtree that will occur unmodified in the result. More precisely, this is the tree consisting of the paths from the root of x to all g gaps in x. Any pointer

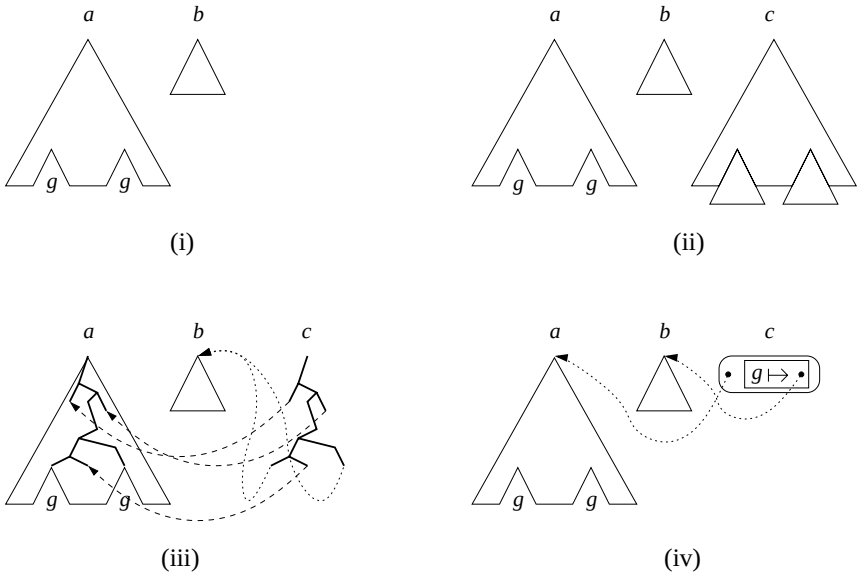

Figure 1: The effect of performing the non-array `plug` operation, c = a.`plug`(g, b). Part (i) shows the two templates, a and b, where a contains two g gaps. Part (ii) shows the naive approach for representing c, where everything has been copied. Part (iii) shows the basic approach from Section 4.2 where only the paths in a that lead to g gaps are copied and new edges are added to the root of b. Part (iv) shows the lazy approach from Section 4.3 where a plug context node is generated for recording the fact that b has been plugged into the g gaps of a. If the structure in (iv) is later traversed completely, the one in (iii) is obtained.

that branches out of these paths in the result points back to the corresponding subtree of x. The ends of the paths, that is, the places where the g gaps of x are, point to the root of y. The y structure is never copied. Note that, in general, this operation will create a DAG rather than a tree, since multiple occurrences of g in x will result in multiple pointers from the result to the root of y. This operation is depicted in Part (iii) of Figure 1. The array `plug` operation is performed similarly, except that the path end pointers point to distinct templates. The `close` operation duplicates the paths to all gaps and removes the gaps from the duplicate.

To be able to find the paths to the g gaps efficiently, we must have additional information in the tree. In each node, we keep a record of the number of occurrences of each gap name in the subtree rooted at that node. Since the occurrence of gaps is usually sparse, this gap presence information can be shared between many nodes and thus will not constitute a large space overhead. Combining this information when constructing new templates is also straightforward. Now, when a `plug` operation into g traverses the tree looking for g gaps, it simply skips all subtrees where the gap presence information indicates that no g gaps exist. This narrows the search down to the paths from the root

to the g gaps. Thus, the execution time for a `plug` operation is proportional to the total number of ancestor nodes of all g gaps in x.

With no *parent* and *previous* pointers, navigation in the singly-linked structure is not as straightforward as in the doubly-linked case. However, since all navigation starts out at the root, we can simply let all navigators remember the traversed path, and then backtrack along this path whenever a backward step is requested. In other words, we let the navigators contain the backward pointers that the XML structure itself omits. Since navigators are always specific to one XML value, we do not restrict sharing by keeping these pointers while the navigator is used. Any navigator step is still performed in constant time, so this additional bookkeeping does not impact the execution time of the algorithm using the navigator.

The `select` operation now simply returns a set of pointers to the nodes pointed to by the navigators resulting from the XPath evaluation. No copying is performed. The total time for performing the `select` operation is proportional to the XPath evaluation time.

The `cut` operation needs to filter out all hits that are descendants of other hits. This can be accomplished by traversing all the navigator paths in parallel, from the bottom up, merging paths as they coincide and throwing away any hit whose path hits the end of another path. If we assume that the XPath evaluator returns its results in document order, this process can be done in time proportional to the total number of ancestors of the hits. Since the XPath evaluator has visited at least all of these nodes to reach the hits, the time used by the XPath evaluation is at least proportional to this total number of ancestors. Thus, the total time for performing the `cut` operation is proportional to the XPath evaluation time.

The `gapify` operation is performed in a manner similar to a `plug` operation, except that the ends of the paths are indicated by the XPath hits, rather than by gaps of a specific name. Instead of navigating downward through the tree using the gap presence information, the `gapify` algorithm navigates upward using the navigator paths. Again assuming that the XPath hits are sorted by document order, this can be done in time proportional to the total number of ancestors of the hits. Thus, the total time for performing the `gapify` operation is proportional to the XPath evaluation time.

So, to summarize, the execution times for the operations will be as follows:

- constructing a tree of size n from its textual representation using the `constant` operation: $O(n + \sum_g \#\text{ancestors}(g))$
- `smash` of k templates: $O(k)$
- `plug` into g: $O(\#\text{ancestors}(g))$
- `close`: $O(\#\text{ancestors of all gaps})$
- `select`, `cut` or `gapify`: $O(\text{XPath evaluation time})$
- converting a template of size n to its textual representation using the `toString` operation: $O(n)$

Regarding memory usage, the operations add only minimally to the memory already used to hold the constant and input templates, since the gap/hit paths that are reconstructed are usually sparse compared to the complete XML trees.

These figures are satisfactory, but we can still do better in some cases, especially when we do not need to traverse the whole result of an operation. This leads us to a further refinement, as explained in the following.

4.3 A lazy data structure

Often, a complete XML transformation will contain several intermediate results that will never be output in their entirety. It may be the case that only parts of these intermediate templates end up in the final result. Or they may even never be output but simply used as operands for further XPath matchings whose results are used in the decision logic of the transformation. For these reasons, the explicit tree construction outlined above is often wasteful, even though it only reconstructs the parts of the result that could not be shared with the operands. What we need is a structure that allows the operations to be performed without any reconstruction taking place until the corresponding parts of the tree are needed by a navigator.

To accomplish this, we introduce special *plug context* nodes in the XML tree, representing a `plug` or `close` operation performed on the subtree. A plug context node has a sequence of children, which are the roots of the left-hand side of the operation. Additionally, it contains a *plug function* which maps a gap name and an index into the XML tree that is plugged into that particular gap. Specifically, if a plug context node has the children $x_1 \ldots x_n$ and the plug function f, then it represents the XML template $x_1 \ldots x_n$ where the ith g gap (in document order, counting from one) is replaced by $f(g, i)$. Part (iv) of Figure 1 illustrates the lazy variant of the plug operation.

The `smash` operation can be performed exactly as before. To perform a `plug` or `close` operation, we simply create a new plug context node and let it point to the roots of the old tree. The plug function is then as follows:

x`.plug(`g`,` y`):`	$\lambda(h,i)$.if $h = g$ then $y$else <[h]>
x`.plug(`g`,` $y_1 \ldots y_k$`):`	$\lambda(h,i)$.if $h = g$ thenif $i \leq k$ then y_i else "" else <[h]>
x`.close():`	$\lambda(h,i).\epsilon$

Here, `""` is the empty string, and ϵ is the empty XML sequence. This makes no difference for template gaps, but for an attribute gap, plugging the empty string will result in an attribute whose value is the empty string, whereas the empty XML sequence will remove the attribute gap, as required by the `close` operation. Similarly, in this formalism, plugging a template gap into an attribute gap of the same name will preserve the attribute gap.

To iterate in this structure, we need to push the plug context nodes further down the tree as we go, so that the node at which we want our navigator to point is always represented directly by a concrete XML node, i.e. not a plug context node.

We refer to this process of pushing down plug context nodes as *normalization*. Let $x_1 \ldots x_n$ be a sequence of XML nodes. This sequence is said to be *normalized* if it is either empty or x_1 is a concrete node. Suppose we have a mechanism for transforming an unnormalized sequence into a normalized one representing the same XML tree. Then we can build a simple navigator (supporting just the *first child*, *first attribute*,

$$\frac{\text{norm}(y_1 \ldots y_m) = \epsilon}{\text{norm}(\{y_1 \ldots y_m, c\}x_2 \ldots x_n) = \text{norm}(x_2 \ldots x_n)}$$

$$\frac{\text{norm}(y_1 \ldots y_m) = z_1 z_2 \ldots z_l}{\text{norm}(\{y_1 \ldots y_m, c\}x_2 \ldots x_n) = \text{norm}(\text{apply}(z_1, c)\{z_2 \ldots z_l, c\backslash z_1\}x_2 \ldots x_n)}$$

$$\text{apply}(\texttt{"}text\texttt{"}, c) = \texttt{"}text\texttt{"}$$

$$\text{apply}(\texttt{<[}g\texttt{]>}, c) = c(g, 1)$$

$$\text{apply}(\texttt{<}e\ \ a_1 \ldots a_k\texttt{>}x_1 \ldots x_n\texttt{</}e\texttt{>}, c) = \texttt{<}e\ \ \{a_1 \ldots a_k, c\}\texttt{>}\{x_1 \ldots x_n, c\backslash a_1 \ldots a_k\}\texttt{</}e\texttt{>}$$

$$\text{apply}(name\texttt{="}value\texttt{"}, c) = name\texttt{="}value\texttt{"}$$

$$\frac{c(g, 1) = \texttt{"}text\texttt{"}}{\text{apply}(name\texttt{=[}g\texttt{]}, c) = name\texttt{="}text\texttt{"}}$$

$$\frac{c(g, 1) = \epsilon}{\text{apply}(name\texttt{=[}g\texttt{]}, c) = \epsilon}$$

$$\frac{c(g, 1) = \texttt{<[}g\texttt{]>}}{\text{apply}(name\texttt{=[}g\texttt{]}, c) = name\texttt{=[}g\texttt{]}}$$

Figure 2: The normalization process. $\{x_1 \ldots x_n, c\}$ denotes a plug context node with children $x_1 \ldots x_n$ and plug function c, and $\texttt{"}text\texttt{"}$ denotes a chardata node with content $text$.

next sibling and *next attribute* operations) on top of this mechanism by transforming sequences of successor (or children) nodes into normalized sequences incrementally as we traverse the tree. We can then build a full-featured navigator on top of this simple navigator in the same manner as for the plain, singly-linked structure. The `select`, `cut` and `gapify` operations can now be implemented exactly as before.

When the plug context nodes are pushed down the tree during the normalization process, the plug functions contained in them change. Since the gap index given to a plug function refers to a global position in the XML template, the plug context for a portion of the template will need to account for all the gaps that precede this portion. More precisely, if the plug context for a sequence of nodes $x_1 \ldots x_n$ is c, then the plug context for a subsequence $x_i \ldots x_j$ will differ from c in a way depending on the presence of gaps in $x_1 \ldots x_{i-1}$. This new context, which we will denote by $c\backslash x_1 \ldots x_{i-1}$, is given by the function $\lambda(h, i).c(h, i + \text{gp}(x_1 \ldots x_{i-1}, h))$, where $\text{gp}(x_1 \ldots x_{i-1}, h)$ is the number of h gaps in $x_1 \ldots x_{i-1}$. This number is available through the gap presence information in the tree.

The normalization process is shown in Figure 2. Normalization proceeds recursively by applying the context to the first of its normalized children and putting the rest of its children into a new plug context node. This application of the context is where the actual context evaluation takes place. The key cases here are the application on a template gap, $\texttt{<[}g\texttt{]>}$, or attribute gap, $name\texttt{=[}g\texttt{]}$, where the plug function is used. Note also how the context update mechanism, $c\backslash x_1 \ldots x_n$, is used to skip the gaps of the first component in a sequence or the attributes in an element.

Normalization, as described in Figure 2, pushes all plug context nodes through every branch of the tree. This is wasteful, since a plug context node will have no effect if none of the gap names it covers (that is, those mapping to anything but the gap itself) occur in the subtree covered by the plug context node. In the actual implementation, a plug context node is only created if it has any effect. Otherwise, its contents are used directly instead. This ensures that the part of the tree through which a particular plug context node is pushed is exactly the ancestors of the involved gaps.

When a normalization has been performed, the internal state of the XML template representation of the normalized template is updated to point to the normalized version. Thus, any pushing down of a plug context node is done at most once. When a template has been traversed completely, its representation has essentially changed into the plain version.

In the plain singly-linked structure, computation of the gap presence information is trivial. The gap presence of any node is simply the sum of the gap presences of its children. This is not so when plug context nodes are present. Fortunately, the gap presence of a context node can be calculated from the gap presences of its children and the targets of its plug function, using the formula

$$\mathrm{gp}(\{x_1 \ldots x_n, c\}, h) = \sum_{g} \sum_{i=1}^{\mathrm{gp}(x_1 \ldots x_n, g)} \mathrm{gp}(c(g, i), h).$$

The actual quantities in the gap presence do not have to be calculated for every operation. The gap presence is represented in a lazy manner, where the count for a particular gap is not calculated until this count is specifically asked for by the navigation algorithm.

Let us now compare the efficiency of this new representation to the plain one. The `constant` and `smash` operators are of course exactly as before. The non-array `plug` and `close` operations take constant time. The array `plug` operation takes time proportional to the number of right-hand-side templates. The time used by the `select`, `cut` and `gapify` operations are still proportional to the time used by the XPath evaluation. Similarly, the time used by `toString` is proportional to the time used to traverse the template. However, because of the plug context nodes that need to be pushed down through the tree, the navigation steps can no longer be performed in constant time. In the worst case, a plug context node will be pushed through all ancestors of the involved gaps, but because of the internal updating, each push will be performed at most once. Since a single push takes constant time, the extra time used by this pushing during traversal is exactly the sum over all operations performed on the template of the time used by that operation in the plain implementation. For this reason, the worst-case amortized execution times for the lazy implementation are identical to the execution times stated for the plain implementation. However, the lazy implementation has the ability to perform its work on a demand basis, which can lead to great savings in practice.

4.4 Java issues

One of the prominent features of immutable, or functional-style, data manipulation is that it works fluently in a multi-threaded environment. For this to work properly in the Java implementation, care must be taken when the internal state of a representation changes. This happens when the result of a normalization replaces the plug context node, and when the gap presence for a particular gap is queried and calculated. These situations are of course properly synchronized in the implementation so that no thread will see the data structure in an inconsistent state.

A ubiquitous Java feature is the ability to compare objects using the `equals` method. This is easily (albeit not very efficiently) done for XML templates by a simple, parallel, recursive traversal. However, to conform to the Java guidelines, any implementation of `equals` must be consistent with the corresponding implementation of the `hashCode` method. Specifically, two identical objects (according to the `equals` method) must have identical hash codes. A (non-trivial) hash code for an XML template must thus reflect the entire XML tree. It would seem that maintaining such a hash code for the result of a `plug` operation is a costly affair. However, if the hash function is chosen such that it is associative *and* commutative with respect to concatenation of XML data, the hash code for the result of a `plug` operation can be calculated from just the hash codes and gap presence information of its constituents. This also enables a more efficient implementation of `equals`: Whenever two compared subtemplates have different hash codes, their equality can be rejected right away. Furthermore, whenever two subtemplates originate from the same original subtemplate unmodified, their object identity verifies their equality.

5 Conclusion

We have presented an overview of the XACT language, focusing on the runtime system. The design of XACT provides high-level primitives for programming XML transformations in the context of a general-purpose language, and, as shown in [16], it permits a precise static analysis. A special feature of the design is that the data-type is immutable, which at the same time is convenient to the programmer and a necessity for precise analysis. However, it also makes it nontrivial to construct a runtime system that efficiently supports all the XACT operations, which is the problem being attacked in this paper.

Our prototype implementation, which consists of the runtime system and the static analyzer supporting the full Java language, is available on the XACT home page: `http://www.brics.dk/Xact/`. Our future work will involve experiments with the prototype implementation to investigate our conjecture that the data structure is sufficiently efficient to be useful in practice.

Also, we plan to integrate XACT into the JWIG system for developing Web services [8]. In such services, XML transformations occur frequently both in the underlying XML databases and in the communication with other programs, such as browsers or other Web services. High-level and efficient approaches for developing Web services together with the ability of obtaining static guarantees of validity of the output

are becoming increasingly important.

References

[1] David Atkins, Thomas Ball, Glenn Bruns, and Kenneth Cox. Mawl: a domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, May/June 1999.

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

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

[4] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, pages 221–231, June 2001.

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

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

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

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

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

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

[11] Vladimir Gapayev and Benjamin C. Pierce. Regular object types. In *Proc. 17th European Conference on Object-Oriented Programming, ECOOP'03*, volume 2743 of *LNCS*. Springer-Verlag, July 2003.

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

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

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

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

[16] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. Technical Report RS-03-19, BRICS, May 2003. Submitted for journal publication.

[17] David A. Ladd and J. Christopher Ramming. Programming the Web: An application-oriented language for hypermedia services. *World Wide Web Journal*, 1(1), January 1996. O'Reilly & Associates. Proc. 4th International World Wide Web Conference, WWW4.

[18] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science, Proc. 1999 Oxford–Microsoft Symposium in Honour of Sir Tony Hoare*, pages 303–321. Palgrave, November 2000.

[19] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

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

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

Recent BRICS Report Series Publications

RS-03-29 **Aske Simon Christensen, Christian Kirkegaard, and Anders Møller.** ***A Runtime System for XML Transformations in Java*****. October 2003. 15 pp.**

RS-03-28 **Zoltán Ésik and Kim G. Larsen.** ***Regular Languages Definable by Lindström Quantifiers*****. August 2003. 82 pp. This report supersedes the earlier BRICS report RS-02-20.**

RS-03-27 **Luca Aceto, Willem Jan Fokkink, Rob J. van Glabbeek, and Anna Ingólfsdóttir.** ***Nested Semantics over Finite Trees are Equationally Hard*****. August 2003. 31 pp.**

RS-03-26 **Olivier Danvy and Ulrik P. Schultz.** ***Lambda-Lifting in Quadratic Time*****. August 2003. 23 pp. Extended version of a paper appearing in Hu and Rodríguez-Artalejo, editors,** ***Sixth International Symposium on Functional and Logic Programming*****, FLOPS '02 Proceedings, LNCS 2441, 2002, pages 134–151. This report supersedes the earlier BRICS report RS-02-30.**

RS-03-25 **Biernacki Dariusz and Danvy Olivier.** ***From Interpreter to Logic Engine: A Functional Derivation*****. June 2003.**

RS-03-24 **Mads Sig Ager, Olivier Danvy, and Jan Midtgaard.** ***A Functional Correspondence between Call-by-Need Evaluators and Lazy Abstract Machines*****. June 2003. 13 pp.**

RS-03-23 **Korovin Margarita.** ***Recent Advances in Σ-Definability over Continuous Data Types*****. June 2003. 26 pp.**

RS-03-22 **Ivan B. Damgård and Mads J. Jurik.** ***Scalable Key-Escrow*****. May 2003. 15 pp.**

RS-03-21 **Ulrich Kohlenbach.** ***Some Logical Metatheorems with Applications in Functional Analysis*****. May 2003. 55 pp. Slighly revised and extended version to appear in** ***Transactions of the American Mathematical Society*****.**

RS-03-20 **Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde.** ***Fast Partial Evaluation of Pattern Matching in Strings*****. May 2003. 16 pp. Final version to appear in Leuschel, editor,** ***ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*****, PEPM '03 Proceedings, 2003. This report supersedes the earlier BRICS report RS-03-11.**