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 documents. Some work in the context of a general-purpose programming language; for example, JDOM [17], 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 defining the transformations. Another approach is to use domain-specific languages, such as XSLT [7], 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 such XML fragments, and a static analysis for detecting programming errors related to the XML operations.

The XML fragments in XACT are immutable for two reasons: First, immutability is always a judicious design choice (*"I would use an immutable whenever I can"*, James Gosling [26]); and second, immutability is a necessity for devising precise and efficient static analyses, in particular, of validity of dynamically constructed XML documents relative to the DTD schemas. The XACT system consists of a simple preprocessor, a runtime library, and a program analyzer. The main contribution of this paper is the

[⋆] Supported by the Carlsberg Foundation contract number ANS-1507/20.
[⋆⋆] Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

description of the XACT runtime system. We present a suitable runtime representation for XML templates that efficiently supports the operations in the XACT API. This is nontrivial mainly because of the immutability of the data type. The companion paper [20] 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 [20] about providing static guarantees for XML transformations written in XACT. Section 4 presents our runtime system and discusses time complexity of the operations. Finally, in Section 5, we evaluate the system by a number of experiments.

Related work The most closely related work is that on JDOM [17], XSLT [18], XQuery [4], XDuce [16], Xtatic [10], ℂDuce [2], XOBE [19], XJ [14], Xen [22], and HaXml [27]. In comparison, the XACT language is based on a combination of the following ideas:

– XACT integrates XML processing into a *general-purpose language*, rather than being a domain-specific language as XSLT or XQuery.
– It applies a *template-based* paradigm for constructing XML values (reminiscent of that in XSLT but unlike the other systems mentioned above).
– XML values are *immutable* (in stark contrast to JDOM, XJ, and Xen).
– Deconstruction of XML values is based on the XPath language [8] (which is also used for similar purposes in XSLT, XQuery, XJ, and optionally also in JDOM).
– Static guarantees are provided through *data-flow analysis*, thereby avoiding the explicit type annotations that are required in approaches based on type systems. Such explicit types can be cumbersome to write and read, and, as noted in [14], explicit types for XML values can be too rigid since the individual steps in a sequence of operations may temporarily invalidate the data unless permitting only bottom-up construction. (JDOM and XSLT provide no similar static guarantees, and the remaining alternatives mentioned above use type systems.)

We refer to the paper [20] for a comprehensive survey of the relation between the language design of XACT and other systems. In the present paper, we focus on the relation to the runtime model of a few representative alternatives: (1) JDOM is generally considered an efficient but rather low-level platform for manipulating XML documents in Java. It provides an explicit tree representation of XML documents where nodes include parent pointers, which permits upwards traversal but prohibits sharing. (2) XSLT is a widely used XML transformation language and many implementations exist. A central part of XSLT is the use of XPath for selection and pattern matching, and much effort has been put into optimizing XPath processors for use in XSLT and other systems [12]. Our implementation of XACT uses an off-the-shelf XPath processor [21] and can hence benefit directly from such work. (3) Both Xtatic and ℂDuce inherit their key features—tree processing in a declarative style with regular types and patterns—from XDuce. Xtatic works in the context of C# whereas ℂDuce is a functional language. The paper [11] describes runtime representations for Xtatic, where the main challenges are immutability (as for XACT), efficient pattern matching (where we apply XPath instead), and DOM interoperability (using techniques that we could also apply). Since no implementation of Xtatic has been available to us, we choose the tuned implementation of ℂDuce as a representative for these systems for quantitative comparisons.

2 The XACT Language

Compared to other XML transformation languages, XACT is designed to be a small sublanguage that can be described in just a few pages. 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 can apply XACT operations while utilizing Java libraries, for example, the sorting facilities, string manipulations, and HTTP communication. We choose to build upon Java because it is widely used and a good representative for the capabilities of modern general-purpose programming languages. Additionally, it is often used as a foundation for Web services, using for example Servlets or SOAP, which involve dynamic construction of XHTML documents or manipulation of SOAP messages.

We build XML documents from *templates* as known from the JWIG language [6]. This approach originates from MAWL [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 has been to extend the formalism to general XML transformations where both construction and deconstruction are 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 core notation for templates is given by xml in the following grammar:

$xml :=$	str	(character data)
$\mid$	`<`$name\ atts$`>`xml`</`$name$`>`	(element)
$\mid$	`<[`g`]>`	(template gap)
$\mid$	$xml\ xml$	(template sequencing)
$atts :=$	$name$`="`$value$`"`	(attribute)
$\mid$	$name$`=[`g`]`	(attribute gap)
$\mid$	ϵ	(empty sequence)
$\mid$	$atts\ atts$	(attribute sequencing)

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. As an example, the following XML template, which can be useful when constructing XHTML documents, contains two template gaps named `TITLE` and `MAIN` and one attribute gap named `COL`:

```
<html>
  <head><title><[TITLE]></title></head>
  <body bgcolor=[COL]><[MAIN]></body>
</html>
```

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

`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 gapify(XPath` p`, Gap` g`)`	– replaces all subtemplates hit by p by g gaps
`XML close()`	– returns this template with all gaps removed
`XML cast(DTD` d`)`	– runtime check for validity
`XML analyze(DTD` d`)`	– compile-time check for validity
`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 a non-constant string

Table 1. The central methods in the `XML` class of XACT.

many times. Gaps can be plugged in any order; construction is not restricted to be bottom-up, in contrast to, for example, XDuce and XOBE.

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 [8]. 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 pure functional languages. In this model, XML templates are unchangeable values and operations have no side-effects. 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 safely return the same instances multiple times, and thread-safety is guaranteed [3]. 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. Furthermore, immutability is a necessity for useful analysis, as described in Section 3.

The immutable Java class `XML`, which represents templates, has the methods shown in Table 1. All parameters of type `Gap`, `XPath`, and `DTD` are assumed to be constants and may be written as strings. The `DTD` parameters are URIs of DTDs.

XACT distinguishes between two different sources of XML data: constant templates and input data. Constant templates are part of the transformation program and are constructed using the `constant` method. The syntax for these templates is the one given by the grammar above. Input to the program is read using the `get` method, which constructs a gap-free template from a non-constant string and checks the result for validity with respect to the given DTD schema. Output from the transformation is achieved through the `toString` method, which returns the string representation of the XML template.

Templates can be combined 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` and `gapify` methods. Both methods take an XPath expression as parameter, which on evaluation returns a set of nodes within the given template[1]. Invoking the `select` method gives an array containing all the subtemplates rooted at nodes in the XPath evaluation result. 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.

The `close` method eliminates all gaps in a template and is commonly used in combination with `toString`. 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[2]. The `equals` method determines equality of `XML` instances, and the `hashCode` method returns a consistent hash code for an `XML` instance. The ability to compare entire XML templates for equality permits templates to be stored in containers as values rather than as objects and can also be useful in the decision logic of transformations. In comparison, other systems either do not have an equality primitive or compare by reference instead of by value.

By placing special `analyze` methods in the code, the compile-time analyzer can be instructed to check for validity relative to the given DTDs. This is usually used in connection with the `toString` method to analyze validity of the output data. Additionally, runtime validation of a template according to a given DTD schema is provided by the `cast` method, which serves a similar purpose for the XACT analysis as the usual cast operation does for the type system of Java.

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. Also, a number of useful macros, presented in [20], for commonly occurring tasks are provided. For example, the `delete` macro effectively deletes the subtrees addressed by an XPath expression by performing a `gapify`

[1] All XPath axes are supported by XACT. Although the paper [20] focuses on the downwards axes, the program analyzer is capable of handling all axes.

[2] The paper [20] describes a more powerful operation `group` and defines `smash` as syntactic sugar. We now treat `smash` as the primitive and express `group` in terms of `smash`, `select`, and `equals` instead.

operation with a fresh gap name. Our implementation also contains a mechanism for declaring XML namespaces for constant templates and XPath expressions.

Example We now consider a simple example, originating from [15], where an address book is filtered in order to produce a phone list. An address book here consists of an `addrbook` root element that contains 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 that contains 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 can be 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);
}
```

We use the `select` operation to build an array of all `person` elements that have a `tel` child. Then, the array entries are combined into a single template, all `addr` elements are deleted, and the result is wrapped into a `phonelist` element[3].

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()")).toString();
  String s2 = XML.smash(x2.select("/person/name/text()")).toString();
  return s1.compareTo(s2);
}
```

The XACT operations here simply extract the character data to be used in the comparison. The phone list can then be sorted by inserting the following line into the `phonelist` method (after the `select` operation):

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

The example shows how XACT integrates XML processing into Java and how a nontrivial transformation task can be intuitive to express using XACT. More example programs can be found at `http://www.brics.dk/Xact/`.

3 Static Guarantees

Transforming data from one XML language to another can be a quite intricate task, even if a high-level programming language is being used. In particular, it can be difficult to ensure at compile-time that the output is always valid with respect to a given

[3] With the notion of *code gaps*, which is included in the syntactic sugar mentioned in [20], the last operation can be written more concisely:
`return [[<phonelist><{list}></phonelist>]];`

DTD schema. A special property of the design of XACT is that it enables precise static analysis for guaranteeing absence of certain programming errors related to XML document manipulation. In the companion paper [20], 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 XML template at this point is guaranteed to be valid relative to the 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` or `gapify` operation will never address any nodes.

Notice that XACT, in contrast to other XML transformation systems that permit static guarantees, does not require every XML variable to be explicitly typed with schema information.

The crucial property of XACT that makes the analysis feasible is that the XML templates are immutable. Analyzing programs that manipulate mutable data structures is known to be difficult [24,23], and the absence of side-effects means that we do not have to model the complex aliasing relations that otherwise may arise.

The analysis is conservative in the sense that it never misses an error, but it might report false errors. Our experiments in [20] 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 in the form of a Java library addressing this issue.

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, we must build the structure representing the template.
- *Combination*: The `plug`, `close`, and `smash` 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, and evaluating an XPath expression on a template all require means for traversing the XML data in various ways. In general, we need 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. We assume that this is sufficient for the XPath engine being used (for example, Jaxen [21] satisfies this).

– *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` and `gapify` operations.

A naive data structure that trivially supports all of these operations is an explicit XML tree with *next_sibling*, *previous_sibling*, *first_child* and *parent* pointers in all nodes (where we encode attributes in the contents sequences). 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 on such a structure would thus be at least linear in the size of the result for each operation. As we show in the following, we can do better using a specialized data structure.

4.1 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 binary tree, that is, a tree with only *first_child* and *next_sibling* pointers but without the *parent* and *previous_sibling* 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.

Recall that, unlike complete XML documents, an XML template does not necessarily have a single root element; rather, it can have an arbitrary sequence of elements, character data and template gaps, which we will refer to as the *top-level nodes*.

To perform a non-array `plug` operation, a`.plug(`g`, `b`)`, we copy just the portion of a 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 a to all g gaps in a. Any pointer that branches out of these paths in the result points back to the corresponding subtree of a. If the gap has a next sibling, we will also need to copy the top-level nodes of b, since the list of successors for these nodes changes. This representation is depicted in Part (iii) of Figure 1. Note that, in general, this operation will create a DAG rather than a tree, since multiple occurrences of g in a will result in multiple pointers from the result to the root of b. 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 graph. In each node, we keep a record of which gap names occur in the subtree represented by that node. Since typical templates contain only few distinct gap names, this *gap presence* information can often be shared between many nodes and will not constitute a large overhead. Combining this information when constructing new templates is also straightforward. Now, when a `plug` operation into g traverses the

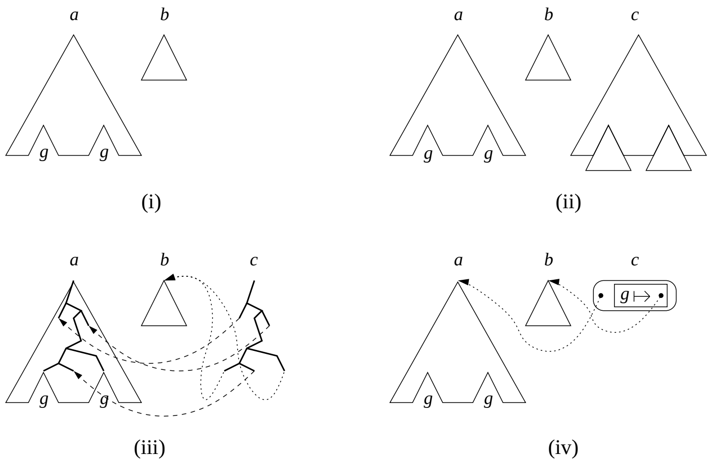

Fig. 1. The effect of performing the non-array `plug` operation, c = a`.plug(`g`,` b`)`. Part (i) shows the two templates, a and b, where a contains two g gaps. Part (ii) shows the naive approach for representing c, where everything has been copied. Part (iii) shows the basic approach from Section 4.1 where only the paths in a that lead to g gaps are copied and new edges are added pointing to the root of b. Part (iv) shows the lazy approach from Section 4.2 where a plug node is generated for recording the fact that b has been plugged into the g gaps of a. When the structure in (iv) is later normalized, the one in (iii) is obtained.

graph looking for g gaps, it simply skips any branch 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 number of nodes that are ancestors of g gaps in a (including preceding siblings because of our use of *first_child* and *next_sibling* pointers), plus the number of top-level nodes in b times the number of g gaps in a. For the array `plug` operation, the last term simply becomes the total number of top-level nodes in the plugged templates.

Constructing the representation of a tree from its textual representation using the `constant` operation takes time proportional to the size of the tree plus, for each node, the number of different gap names that appear in its subtree. The time for converting a template to text using `toString` is proportional to the template size.

Navigation in this structure is not as straightforward as in the doubly-linked case, since navigating backwards with *parent* or *previous sibling* requires information that is not available in the tree. We can support these directions by letting the navigators remember the entire path back to the root, and then backtrack along this path whenever a backward step is requested. In other words, we let the navigators contain all 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. Taking any navigator step still takes constant time.

The `select` operation simply returns a set of pointers to the nodes pointed to by the navigators that result from the XPath evaluation. Only the nodes pointed to are copied to make sure that their *next_sibling* pointers are empty. The total time for performing the `select` operation is proportional to the XPath evaluation time. The `gapify` operation first evaluates the XPath expression, resulting in a set of navigators that represent the addressed nodes. The tree from the root to these nodes is then copied, as for the `plug` operation. After that, the gap information in the nodes of the new tree is updated in a bottom-up traversal to include the new gaps. The total time for performing the `gapify` operation is proportional to the XPath evaluation time, which dominates the time for the other steps. Finally, the `smash` operation can be simulated by making a sequence of gap nodes with a fresh gap name and performing an array plug operation into these. This takes time proportional to the total number of top-level nodes in the smashed templates.

These figures may seem satisfactory; however, it turns out that this approach has some drawbacks as the following observations reveal.

– In a sequence of plug operations, each individual plug may create many nodes that will be replaced in a subsequent plug operation. If the intermediate results are not needed except as arguments for the subsequent plug operations (which is usually the case), constructing these nodes is unnecessary and wasteful. For example, a common idiom is to use a template `<li><[item]></li><[more]>` to build a list of `li` elements by repeatedly plugging the template itself into the `more` gap. Such a construction would take quadratic time in the length of the constructed list, since all preceding siblings need to be copied each time.
– Traversing the structure recursively when looking for gaps can lead to unwieldy stack sizes, since the ancestor nodes of the gaps include all preceding siblings. This problem clearly shows up in practice—the algorithm is unable to handle XML documents with more than a few thousand mutual siblings.

These observations lead us to a further refinement, as explained in the following section.

4.2 A lazy data structure

We now present a modification of the basic structure that allows the operations to be performed lazily without any reconstruction taking place until explicit traversal of the tree is required. This effectively groups plug operations together in a way that permits list structures to be built in linear time.

To accomplish this, we introduce special *operation nodes* in the graph, each representing a `plug` or `close` operation (with `smash` being simulated by array `plug` as before). We call all other nodes *concrete nodes*. An operation node has one designated child node, which represents the *this* operand. There are three variants of operation nodes, corresponding to the two variants of the `plug` operation and the `close` operation, respectively: the *non-array plug node* is labeled with a gap name and has one extra edge corresponding to the value being plugged in; similarly, the *array plug node* is labeled with a gap name and an array of extra edges; and the *close node* has no extra information. Intuitively, an operation node merely records the fact that a `plug` or `close` operation has occurred without actually performing it. Part (iv) of Figure 1 illustrates this lazy variant of the plug operation.

As long as only `plug`, `close` and `smash` operations are performed, the resulting template will be represented by a DAG of operation nodes and concrete nodes, where all ancestors of operation nodes are themselves operation nodes. When the actual tree is needed, we need to unfold this structure into a DAG of only concrete nodes, so that the previously described navigation mechanism can be used. We refer to this process of eliminating operation nodes as *normalization*.

To perform normalization, we traverse the DAG depth-first while keeping track of the current *plug context*. A plug context is a map from gap names to nodes, defined by a list of operation nodes. The current plug context is always defined by the list of ancestor operation nodes of the current node. A non-array plug node maps its corresponding gap name to the root of the plugged template; an array plug node initially maps its gap name to the root of the first plugged template; and a close node maps all gap names to a special value *remove*. When more than one operation node mapping from the same gap name exist among the ancestors, the one furthest down in the DAG has precedence as it corresponds to an earlier plug operation.

Whenever we encounter a gap node with a name for which there is a mapping in the current plug context, we recursively traverse the template rooted at the node targeted by the mapping—or remove the gap node in case of the value *remove*. If the operation node is an array plug node, its mapping is changed to the next node on its list, or to the empty template if the list is exhausted. Note that the plug context in this traversal of the plugged template is defined by the operation node ancestors in the complete DAG, which includes the ones in the plugged template plus the ones created after it was plugged.

Just as for the plug operation in the basic approach, we only traverse the part of the DAG which actually needs to be duplicated. That is, we skip any branch where the gap presence information indicates that no gap exist for which there is a mapping in the current plug context. The only exception to this is the top-level nodes of plugged templates, which in general need to be duplicated, as before.

Following this strategy, we essentially perform a single traversal of the part of the final result which could not be shared with any of the constituent templates. Thus, assuming that the context lookup operations can be performed in amortized constant time (which can be accomplished by caching lookups), and assuming a constant number of distinct gap names, the running time for the entire normalization process is proportional to the number of newly created nodes. Since this is bounded by the size of the result and the result is typically traversed completely anyway, this is a satisfactory result.

To alleviate the stack requirement of the traversal, we use pointer reversal [25], which in essence uses the newly generated nodes as an explicit recursion stack. The recursion involved in the recursive unfolding of plugged templates mentioned above is done using a separate, explicit stack. Thus, with this strategy, the call stack usage is bounded by a constant, and the overall memory requirements are significantly reduced, compared to the purely recursive approach.

4.3 Java issues

One of the prominent features of immutable 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 operation nodes—and this is of course properly synchronized in the implementation so that no thread will see the data structure in an inconsistent state, and no two threads will perform the same normalization simultaneously. Note also that the pointer reversal only changes newly created nodes, so another thread can traverse (and even normalize) a template sharing parts with the one being normalized without causing any problems.

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. To conform to the Java guidelines, any implementation of `equals` must be consistent with the corresponding implementation of the `hashCode` method. To provide this consistency, each node includes a hash code representing the XML tree rooted at the node (including following siblings). The hash code for the entire template is then the hash code of the leftmost top-level node. 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 Evaluation

This section describes experiments with our prototype implementation of the XACT runtime system. The main goal is to gather runtime performance measurements for a range of typical XML transformations in order to compare the performance of XACT with that of related systems. Due to the limited space, we can only provide a brief report on our evaluation results.

We have collected a suite of benchmark programs, most of which are inspired by XML transformations developed in other languages. A few programs have been developed to specifically test the worst-case behavior of our implementation. Altogether the suite covers a broad spectrum of typical XML transformation tasks.

Most of the related technologies mentioned in Section 1 are currently being developed by other research teams. Unfortunately, only a few have wished to provide an implementation, making it impossible to do a complete performance comparison of all the systems. Instead we have picked JDOM, XSLT and ℂDuce—for which optimized runtime systems are available—as good representatives for the different approaches. The JDOM and ℂDuce measurements are obtained using the latest releases (JDOM Beta 10, and ℂDuce 0.1.1.) The XSLT measurements are obtained using Apache Xalan 2.6, which supports the complete XSLT 1.0 language and is among the fastest Java-based implementations. For XACT, we use the lazy approach described in Section 4.2. All experiments have been executed on an 3.0 GHz Intel Pentium 4 machine with 1 GB RAM running Red Hat Linux 9.0 with Sun's Java 2 SE 1.4.2 and O'Caml 3.0.7. Since the focus of this paper is runtime performance we do not measure compilation and type checking. Furthermore, the price of parsing input XML documents says little about the relative strengths of the implementations, so this cost is excluded from measurements in order to give a fair comparison.

We start by comparing XACT with XSLT using four typical XML transformation tasks. Two transformations originate from the XSLTMark benchmark suite [9]: `Backwards` mirrors its input document by reversing the order of all node sequences; `DBOnerow` queries a person database for a single entry and transforms it into XHTML. Performance on mixed content documents is compared by `Uppercase`, which transforms all names in an address book into uppercase characters. Phonelist is the example from Section 2 transforming an address book into a sorted phone list. The transformations are executed on input XML documents of size 100 KB, 1 MB, and 10 MB.

	100 KB		1 MB		10 MB	
	XSLT	XACT	XSLT	XACT	XSLT	XACT
Backwards	551 ms	421 ms	1,615 ms	1,513 ms	15,373 ms	11,599 ms
DBOnerow	279 ms	160 ms	754 ms	274 ms	4,048 ms	994 ms
Uppercase	431 ms	246 ms	1,234 ms	634 ms	8,810 ms	5,365 ms
Phonelist	494 ms	423 ms	1,351 ms	1,799 ms	8,029 ms	21,834 ms

These figures indicate that the performances of the two are roughly similar. The main benefits of XACT compared to XSLT are the static guarantees and the possibility of applying the full Java language. For example, the Uppercase benchmark is only expressible in XSLT because this language contains a built-in function (translate) for mapping individual characters to other characters; more advanced character data transformations are not possible in XSLT without implementation dependent extension functions.

Next, we compare XACT with JDOM using the Linkset transformation (Example 15.8 in [13]), which extracts a set of links from an RDF feed, and the Phonelist transformation, which is described above.

	100 KB		1 MB		10 MB	
	JDOM	XACT	JDOM	XACT	JDOM	XACT
Linkset	23 ms	146 ms	128 ms	316 ms	304 ms	1,837 ms
Phonelist	80 ms	422 ms	408 ms	1,799 ms	3,212 ms	21,834 ms

These experiments indicate that the JDOM approach with mutable tree updates and purely navigational access, as one would expect, performs better than the immutable XACT approach based on XPath. However, this should be contrasted by the fact that the the XACT transformations are both shorter and more readable than the JDOM transformations. Furthermore, the XACT transformations are statically type safe in contrast to those written with JDOM.

For the comparison of XACT and ℂDuce we use our Phonelist transformation and the Split transformation, which is a benchmark program developed by the ℂDuce team and used in their performance comparisons.

	100 KB		1 MB		10 MB	
	ℂDuce	XACT	ℂDuce	XACT	ℂDuce	XACT
Phonelist	156 ms	422 ms	1,747 ms	1,799 ms	21,579 ms	21,834 ms
Split	94 ms	496 ms	496 ms	1,729 ms	*error*	12,897 ms

Since XACT uses Java and ℂDuce uses O'Caml, the performance is difficult to compare[4], but on these few benchmarks there seems to be no significant time difference for larger data sets. When running the ℂDuce Split transformation on the 10MB document, it runs out of memory, indicating that that the internal XML representation in XACT is more compact than the one in ℂDuce.

[4] To exclude parsing time for ℂDuce, we measured the full time including parsing and then subtracted the time for performing the identity transformation.

To demonstrate that the lazy approach is preferable to the basic one, we compare the two using a benchmark `Logging`, which extracts statistical information from a web server log file and exhibits the quadratic blowup for the basic approach:

	XACT (basic)	XACT (lazy)
`Logging` (100 KB)	709 ms	639 ms
`Logging` (1 MB)	3,189 ms	1,926 ms
`Logging` (3 MB)	11,227 ms	3,836 ms
`Logging` (10 MB)	*stack overflow*	9,011 ms

These figures show that the lazy approach can lead to significant saving in practice and how it scales smoothly to large documents.

In general, we conclude that the runtime system is sufficiently efficient. Our goal has not been to outperform the alternative XML transformation systems, but rather to be comparable in runtime performance and scalability, which complements the convenient language design and static analysis that XACT also provides.

Obviously, there are ways to improve performance further. We plan to experiment with caching of XPath parse trees, handling simple XPath expressions without involving the general XPath engine, and compiling XPath expression to basic navigation steps (as also done in the XJ project). Also, we believe that it is possible to exploit the knowledge gained from the static analysis for optimizing the evaluation of XPath expressions.

6 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 [20], 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 main problem being addressed in this paper. Our experiments indicate that the runtime system being proposed is sufficiently efficient to be practically useful.

Our prototype implementation, which consists of the runtime system, the desugarer, and the static analyzer supporting the full Java language, is available on the XACT home page: `http://www.brics.dk/Xact/`.

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. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *Proc. 8th ACM International Conference on Functional Programming, ICFP '03*, August 2003.
3. Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, June 2001.
4. Scott Boag et al. XQuery 1.0: An XML query language, November 2003. W3C Working Draft. `http://www.w3.org/TR/xquery/`.

5. Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.
6. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.
7. James Clark. XSL transformations (XSLT) specification, November 1999. W3C Recommendation. `http://www.w3.org/TR/xslt`.
8. James Clark and Steve DeRose. XML path language, November 1999. W3C Recommendation. `http://www.w3.org/TR/xpath`.
9. DataPower. XSLTMark, 2001. `http://www.datapower.com/xmldev/xsltmark.html`.
10. 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.
11. Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. XML goes native: Run-time representations for Xtatic, 2004.
12. Georg Gottlob, Christoph Koch, and Reinhard Pichler. XPath query evaluation: Improving time and space efficiency. In *Proc. 19th International Conference on Data Engineering, ICDE'03*. IEEE Computer Society, March 2003.
13. Elliotte Rusty Harold. *Processing XML with Java*. Addison-Wesley, November 2002.
14. Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael Burke, Vivek Sarkar, and Rajesh Bordawekar. XJ: Integration of XML processing into Java. Technical Report RC23007, IBM Research, 2003.
15. 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.
16. Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2), 2003.
17. Jason Hunter and Brett McLaughlin. JDOM, 2004. `http://jdom.org/`.
18. Michael Kay. XSL transformations (XSLT) version 2.0, May 2003. W3C Working Draft. `http://www.w3.org/TR/xslt20/`.
19. Martin Kempa and Volker Linnemann. Type checking in XOBE. In *Proc. Datenbanksysteme für Business, Technologie und Web, BTW '03*, volume 26 of *LNI*, February 2003.
20. Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.
21. Bob McWhirter et al. Jaxen, December 2003. `http://jaxen.org/`.
22. Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with rectangles, triangles, and circles. In *Proc. XML Conference and Exposition, XML '03*, December 2003.
23. 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.
24. 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.
25. H. Schorr and W. M. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.
26. Bill Venners. A conversation with James Gosling. JavaWorld, June 2001.
27. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming, ICFP '99*, September 1999.