Type Checking with XML Schema in XACT

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

Abstract

XACT is an extension of Java for making type-safe XML transformations. Unlike other approaches, XACT provides a programming model based on XML templates and XPath together with a type checker based on data-flow analysis.

We show how to extend the data-flow analysis technique used in the XACT system to support XML Schema as type formalism. The technique is able to model advanced features, such as type derivations and overloaded local element declarations, and also datatypes of attribute values and character data. Moreover, we introduce optional type annotations to improve modularity of the type checking.

The resulting system supports a flexible style of programming XML transformations and provides static guarantees of validity of the generated XML data.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.4 [**Software Verification**]: Validation; I.7.2 [**Document and Text Processing**]: Computing Methodologies

General Terms

Languages, Design, Verification

Keywords

XML, XML Schema, Java, language design, static analysis

1 Introduction

The overall goal of the XACT project is to integrate XML into general-purpose programming languages, in particular Java, such that programming of XML transformations can become easier and safer than with the existing approaches. Specifically, we aim for a system that supports a high-level and flexible programming style, permits an efficient runtime model, and has the ability to statically guarantee validity of generated XML data.

In previous papers, see [15, 14], we have presented the first steps of our proposal for a system that fulfills these requirements. Our language, XACT, is an extension of Java where XML fragments can be manipulated through a notion of *XML templates* using XPath for navigation. Static guarantees of validity are provided by a special data-flow analysis that builds on a lattice structure of *summary graphs*.

The existing XACT system has two significant weaknesses: first, it only supports DTD as schema language, and it is generally agreed that DTD has insufficient expressiveness for modern XML applications; second, the data-flow analysis is a whole-program analysis that has poor modularity properties and hence does not scale well to larger programs. In this paper, we present an approach for attacking these issues.

Contributions

We have previously shown a connection between summary graphs and regular expression types [4, 10]. Also, it is known how regular expression types are related to RELAX NG schemas [7] and how schemas written in XML Schema [22, 2] can be translated into equivalent RELAX NG schemas [12]. We exploit these connections in this paper. Our main contributions are the following:

- We present a translation from XML Schema to summary graphs and an algorithm for validating summary graphs relative to schemas written in XML Schema, all via RELAX NG. This provides the foundation for using XML Schema as type formalism in XACT.
- We introduce optional typing in XACT so that XML template variables can be optionally typed with schema constructs (element names and simple or complex types). We show how this can lead to a validity analysis which is more modular, in the sense that it avoids iterating over the whole program.

Together, these improvements effectively remedy the weaknesses mentioned earlier. Furthermore, the results can be seen as indications of the strength of summary graphs and the use of data-flow analysis for validating XML transformations.

As an additional contribution, we identify a subset of RELAX NG that is sufficient for translation from XML Schema and where language inclusion checking is tractable.

Example

The resulting XACT language can be illustrated by a small toy program that uses the new features. This program converts a list of business cards represented in a special XML language into XHTML, considering only the cards where a phone number is present:

*Supported by the Carlsberg Foundation contract number 04-0080.
†Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

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

public class PhoneList {
  static {
    String[] ns =
      {"b", "http://businesscard.org",
       "h", "http://www.w3.org/1999/xhtml",
       "s", "http://www.w3.org/2001/XMLSchema"};
    XML.setNamespaceMap(ns);
  }

  XML<h:html[s:string TITLE, h:Flow MAIN]> wrapper;

  void setWrapper(String color) {
    wrapper =
      [[<h:html>
          <h:head>
            <h:title><[s:string TITLE]></h:title>
          </h:head>
          <h:body bgcolor={color}>
            <h:h1><[s:string TITLE]></h:h1>
            <[h:Flow MAIN]>
          </h:body>
        </h:html>]];
  }

  XML<h:ul> makeList(XML<b:cardlist> x) {
    XML r = [[<h:ul><[CARDS]></h:ul>]];
    XMLIterator i =
        x.select("//b:card[b:phone]").iterator();
    while (i.hasNext()) {
      XML c = i.next();
      r = r.plug("CARDS",
           [[<h:li>
               <h:b><{c.select("b:name/text()")}></h:b>,
               phone: <{c.select("b:phone/text()")}>
             </h:li>
             <[CARDS]>]]);
    }
    return r;
  }

  XML<h:html> transform(String url) {
    XML cardlist = XML.get(url, "b:cardlist");
    setWrapper("white");
    return wrapper.plug("TITLE", "My Phone List")
                  .plug("MAIN", makeList(cardlist));
  }

  public static void main(String[] args) {
    XML<h:html> x = new PhoneList().transform(args[0]);
    System.out.println(x);
  }
}
```

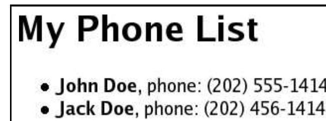
The general syntax for XML template constants and the meaning of the methods `select`, `plug`, `get`, and various others are described further in Section 2.

In the first part of the program, some global namespace declarations are made. Schemas for these namespaces are supplied externally (the schema for the business card XML language is shown in Section 3). Then a field `wrapper` is defined, holding an XML template that must be an `html` tree, potentially with `TITLE` gaps and `MAIN` gaps, which may occur in place of fragments of type `string` and `Flow`, respectively (all of appropriate namespaces). The method `setWrapper` assigns such an XML template to the `wrapper` field. This template has two gaps named `TITLE` and one named `MAIN`. Additionally, it has one code gap where the value of the `color` parameter is inserted. The method `makeList` iterates through a list of `card` elements that have `phone` children and builds an XHTML list. The method `main` loads in an XML document containing a list of business card, invokes the `setWrapper` method, then constructs a complete XHTML document by plugging values into the `TITLE` and `MAIN` gaps using the `makeList` method, and finally outputs this document.

As an example, the program transforms the input

```
<cardlist xmlns="http://businesscard.org">
  <card>
    <name>John Doe</name>
    <email>john.doe@widget.inc</email>
    <phone>(202) 555-1414</phone>
  </card>
  <card>
    <name>Zacharias Doe</name>
    <email>zach@notmail.com</email>
  </card>
  <card>
    <name>Jack Doe</name>
    <email>jack@mailorder.edu</email>
    <email>jack@geemail.com</email>
    <phone>(202) 456-1414</phone>
  </card>
</cardlist>
```

into an XHTML document that looks as follows:



Note that some XML variables in the program are declared by the type `XML`, which represents all possible XML templates, and others use a more constrained type, such as, the declaration of `wrapper` or the signature of `makeList`. XACT now allows the programmer to combine these two approaches. The static type checker uses data-flow analysis to reason about variables that are declared using the former approach, and it conservatively checks that the annotated types are preserved by execution of the program. For this program, one consequence is that the `makeList` method, whose signature is fully annotated, can be type checked separately, and invocations of this method can be type checked without considering its body. (We discuss fields and side-effects in Section 7.) Also note that the type checker can now reason about XML Schema types rather that being limited to DTD.

Related Work

There are numerous other projects having similar goals as XACT; the paper [19] contains a general survey of different approaches. The ones that are most closely related to ours are XJ [9], Cω [1], and XDuce and its descendants [10]. XACT is notably different in two ways: first, although variants of XML templates are widely used in Web application development frameworks, this paradigm is not supported by other type-safe XML transformation languages, which typically allow only bottom-up XML tree construction; second, the annotation overhead is minimal since schema types are only required at input and output, whereas the others require schema type annotations at all XML variable declarations. We believe that both aspects in many cases makes the XACT programming style more flexible. Furthermore, our data-flow analysis also tracks all Java string operations via a separate analysis [6], which enables XACT to reason about validity of attribute values and character data. (In fact, an additional consequence of the extensions

described here is that our static analyzer can also model computed names of elements and attributes.)

With the extensions proposed in this paper, XACT becomes closer to XJ [9], which also uses XML Schema as type formalism and XPath for navigation. Still, our use of *optional* type annotations avoids a problem that can make the XJ type checker too rigid: with mandatory type annotations at all variable declarations in XJ it is impossible to type check a sequence of operations that temporarily invalidates data. The types that are involved in XML transformations are often exceedingly complicated and difficult to write down, and types for intermediate results often do not correspond to named constructs in preexisting schemas. The benefits of type annotations are that they can serve as documentation in the programs and they can lead to faster type checking. By now supporting optional annotations, XACT gets the best from the two worlds.

Moreover, XJ represents XML data as *mutable* trees, which incurs a need for expensive runtime checks to preserve data validity. In XJ, subtyping is nominal, whereas our approach gives semantic (or structural) subtyping. A discussion of subtyping can be found in [8]. Note that although XML Schema does contain mechanisms for declaring subtyping relationships nominally, the choice of supporting XML Schema as type formalism in XACT does not force us to use nominal subtyping. We use schemas only as notation for defining sets of XML values—the internal structure of the notation being used is irrelevant.

The XDuce language family is based on the notion of regular expression types. As mentioned earlier, a connection between regular expression types and a variant of the summary graphs used in our program analysis is shown in [4]. Also, the formal expressiveness of regular expression types and RELAX NG both correspond to that of regular tree languages. We return to these relations in Sections 5 and 6. As XACT, the XTATIC language [8], which is one of the descendants of XDuce, incorporates XML into an object-oriented language in an immutable style.

The Cω language adds XML support to C♯ by combining structural sequences, unions, and products with objects and simple values. The basic features of XML Schema may be encoded in the type system, however little documentation of this is available. Rather than use full XPath for navigation in XML trees as in XACT, Cω uses a reminiscent notion of generalized member access that is closer to ordinary programming notation.

The paper [18] describes a validity analysis for XSLT transformations, which is also based on summary graphs. The techniques we present here for handling XML Schema as type formalism can be transferred seamlessly to that analysis.

The type annotations we introduce are reminiscent of the notion of programmer–designer contracts proposed in [3]. In both cases, static declarations constrain how XML templates may be combined in the programs.

The paper [20] contains a useful classification of schema languages in terms of categories of tree grammars: DTD corresponds to *local tree grammars* where the content model of an element can only depend on the name of the element; XML Schema corresponds to the larger category of *single-type tree grammars* where elements that are siblings and have the same name must have identical content models; and RELAX NG corresponds to the even more general category of *regular tree grammars*, which is equivalent to tree automata. With our new results, XACT supports single-type tree grammars as type formalism.

Overview

In Sections 2 and 3 we begin by briefly recapitulating the design of XACT and RELAX NG, and we characterize a subset of RELAX NG, called *Restricted RELAX NG*, that we will use as an intermediate language in the program analysis. Then, in Section 4 we introduce a variant of summary graphs. In Sections 5 and 6 we explain how schemas written in XML Schema can be converted into summary graphs via Restricted RELAX NG, how to check validity of summary graphs relative to Restricted RELAX NG schemas, and how these results can be used in XACT to provide static guarantees of XML transformations. In Section 7 we introduce optional typing using XML Schema constructs and discuss the resulting language design. Finally, we present our conclusions in Section 8.

Note that we here report on work in progress, and not all of what we present has yet been implemented and tested in practice so we cannot at this stage present experimental results. Also, the limited space prevents us from going into details of our algorithms and of the systems we build upon—instead, this paper aims to present an informal overview of our ideas.

2 The XACT Programming Language

We begin with a brief overview of the XACT language as it looks before adding our new extensions. In XACT, XML data is represented as *templates*, which are well-formed XML fragments that may contain gaps in place of elements or attribute values. A gap is either a name or a piece of code that evaluates to a string or an XML template. As an example, the following XML template contains four gaps: two named `TITLE`, one named `MAIN`, and one containing the expression `color`:

```
<h:html>
  <h:head>
    <h:title><[TITLE]></h:title>
  </h:head>
  <h:body bgcolor={color}>
    <h:h1><[TITLE]></h:h1>
    <[MAIN]>
  </h:body>
</h:html>
```

The special immutable class `XML` corresponds to the set of all possible XML templates. The central operations on this class are the following:

`constant:` a static method that creates a template from a constant string (the syntax `[[foo]]` is sugar for `XML.constant("foo")` where quotes, whitespace, and gaps have been transformed);

`plug:` inserts a given string or template into all gaps of a given name in this template;

`select:` returns the sub-templates of this template that are selected by a given XPath expression;

`get:` a static method that creates a template from a non-constant string and checks (at runtime) that it is valid relative to a given constant schema type;

`cast:` performs a runtime check of validity of this template relative to a given constant schema type;

`analyze:` instructs the static type checker to verify that this template will always be valid relative to a given schema type when the program runs; and

`toString:` converts this template to its textual representation.

A *schema type* is the name of an element (or, with our extension from DTD to XML Schema, a simple type or a complex type) that is declared in a schema. The *language* of a schema type is defined

as the set of XML documents or document fragments that are valid relative to the schema type. Note that in this version of XACT, before incorporating the extensions suggested in this paper, schema types appear only at `get`, `cast`, and `analyze` operations. In particular, declarations use the general type `XML`.

The primary job of the static type checker is to verify that only valid XML data can occur at program locations marked by `analyze` operations, under the assumption that `get` and `cast` operations always succeed. (It also checks properties of `plug` and `select` operations, which is less relevant here.)

3 Defining a Subset of RELAX NG

A RELAX NG schema [7] is essentially a top-down tree automaton that accepts a set of valid XML trees. It is described by a grammar consisting of recursively defined *patterns* of various kinds, including the following: `element` matches one element with a given name and with contents and attributes described by a sub-pattern; `attribute` similarly matches an attribute; `text` matches any character data or attribute value; `group`, `optional`, `zeroOrMore`, `oneOrMore`, and `choice` correspond to concatenation, zero or one occurrence, zero or more occurrences, one or more occurrences, and union, respectively; `empty` matches the empty sequence of nodes; and `notAllowed` corresponds to the empty language. In addition, the pattern `interleave` matches all possible mergings of the sequences that match its sub-patterns.

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

To ensure regularity, there is an important restriction on recursive pattern definitions: recursion is only allowed if passing through an `element` pattern.

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

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

Furthermore, RELAX NG contains various modularization mechanisms, which we can ignore here. As all other type-safe XML transformation languages, we also ignore `ID` and `IDREF` attributes from DTD and the equivalent compatibility features in RELAX NG.

As mentioned in the introduction, we handle XML Schema via a translation to RELAX NG, thus using RELAX NG as a convenient intermediate language that avoids the many complicated technical details of XML Schema. However, we only use a subset of RELAX NG, which we call *Restricted RELAX NG*, being characterized as follows.

First, we define some terminology that we need. We say that a pattern *p top-level-contains* a pattern *q* if *p* and *q* are identical or *p* contains *q* (as a child or further descendant) where contents of `element` and `attribute` patterns are ignored. A *content pattern* is a pattern that top-level contains one or more `element`, `data`, or `text` patterns (or `list` or `value` patterns, which we otherwise ignore here for simplicity). An *attribute list pattern* is a pattern that top-level contains one or more `attribute` patterns.

A Restricted RELAX NG schema satisfies the following syntactic requirements:

[single-type grammar] For every `element` pattern *p*, any two `element` patterns that are top-level-contained by the child of *p* and have non-disjoint name classes must have the same (identical) content. (This requirement limits the notation to single-type grammars.)

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

[interleaved content] Every pattern that has a child that top-level contains an `interleave` content pattern must be a `group` or `element` pattern. Also, a `group` pattern that top-level contains an `interleave` content pattern must have only one content pattern child. (This requirement makes it easier to check inclusion of `interleave` patterns, as explained in Section 6.)

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

Restricted RELAX NG has two important properties: first, it is sufficient for making an exact and simple embedding of XML Schema; second, it makes the summary graph validation in Section 6 more tractable than using XML Schema directly or supporting full RELAX NG.

The following schema written in XML Schema may be used to describe the input to the example program shown in Section 1:

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

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

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

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

</schema>
```

Assuming `cardlist` as root element name, this can be translated into the following Restricted RELAX NG schema (here using the compact RELAX NG syntax):

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

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

The translation from XML Schema to Restricted RELAX NG is exact and the size of the output schema is proportional to the size of the input schema. Most XML Schema constructs map directly to RELAX NG, and we will not here explain the details of the translation. However, a few points are worth mentioning.

First, the `all` construct maps to the `interleave` pattern. Because of the limitations on the use of `all` in XML Schema, this does not violate the [interleaved content] requirement.

Second, we can ignore default declarations since we only care about validation and not of normalization of the input—except that we treat an attribute or content model as optionally absent if a default is declared.

Third, wildcards can be converted into name classes. If `processContents` of an element wildcard is set to `skip`, then we make a recursive pattern that matches any XML tree.

Fourth, the most tricky parts of the translation involve type derivations and substitution groups. Assume that an element e has type t and there exists a type t' that is derived by extension from t. In this case, an occurrence of e must match either t or t', and in the latter case e must have a special attribute `xsi:type` with the value t' (in the former case, the attribute is permitted but not required). We handle this situation by encoding the `xsi:type` information in the element name. More precisely, we create a new element pattern whose name is the name of e followed by the string `%`t' and whose content corresponds to the definition of t'. Each reference to e is then replaced by a choice between e and the variants with extended types. The `xsi:nil` feature is handled similarly. Now assume that another element f has type t' and is declared as in the substitution group of e. This means that f elements are permitted in place of e elements. In Restricted RELAX NG, this is expressed simply by replacing all references to e elements by choices of e and f elements. Again, because of limitations on the `all` construct and the substitution group mechanism in XML Schema, this cannot lead to violations of the [single-type grammar] requirement, nor of the general RELAX NG requirement that `interleave` branches must be disjoint.

By the translation to Restricted RELAX NG, a schema type corresponds to a pattern definition:

- a simple type corresponds to a pattern, which we call a *simple-type pattern*, that can only contain the constructs `data`, `choice`, `list`, and `value`;
- a complex type corresponds to a pattern, which we call a *complex-type pattern*, that consists of a group of two subpatterns—one describing a content model and one describing attributes; and
- an element declaration corresponds to an `element` pattern that contains a simple-type pattern or a complex-type pattern.

We use these observations in Section 6.

4 Summary Graphs in Validity Analysis

The static type checker in XACT works in two steps. First, a data-flow analysis of the whole program is performed, using the standard data-flow analysis framework [11] but with a highly specialized lattice structure where abstract values are *summary graphs*. A summary graph is a finite representation of a potentially infinite set of XML templates, much like a schema but tailor-made for use in the program analysis [15]. Second, when the fixed point has been computed, we check that the sets of templates represented by the resulting summary graphs are valid relative to the respective schemas.

To allow a smooth integration of XML Schema as a replacement for DTD, we slightly modify the definition of summary graphs as explained below and change the summary graph validation algorithm accordingly and to work with Restricted RELAX NG (the old algorithm supported DTD via an embedding into DSD2 [17]).

A summary graph, as it is defined in [15], has two parts: one that is set up for the given program and remains fixed during the iterative data-flow analysis, and one that changes monotonically during the analysis.

The fixed part contains finite sets of nodes of various kinds: element nodes ($N_\mathcal{E}$), attribute nodes ($N_\mathcal{A}$), chardata nodes ($N_\mathcal{C}$), and template nodes ($N_\mathcal{T}$). These node sets are determined by the use of schemas, template constants, and `XML` operations in the program. The former three sets represent the possible elements, attributes, and chardata sequences that may arise when running the program. The template nodes represent sequences of template gaps, which either occur explicitly in template constants or implicitly due to XML operations or schemas. Additionally, the fixed part specifies a number of maps: *name* assigns a name to each element node and attribute node; $attr : N_\mathcal{E} \to 2^{N_\mathcal{A}}$ associates attribute nodes with element nodes; $contents : N_\mathcal{E} \to N_\mathcal{T}$ connects element nodes with descriptions of their contents; and $gaps : N_\mathcal{T} \to G^*$ associates a sequence of gap names from a finite set G with each template node.

The changing part of a summary graph consist of:

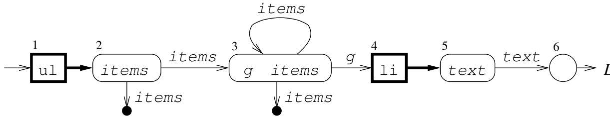
- a set of root nodes $R \subseteq N_\mathcal{E} \cup N_\mathcal{T}$;
- template edges $T \subseteq N_\mathcal{T} \times G \times (N_\mathcal{T} \cup N_\mathcal{E} \cup N_\mathcal{C})$;
- string edges $S : N_\mathcal{C} \cup N_\mathcal{A} \to REG$ where REG are all regular string languages over the Unicode alphabet; and
- a gap presence map $P : G \to 2^{N_\mathcal{A} \cup N_\mathcal{T}} \times 2^{N_\mathcal{A} \cup N_\mathcal{T}} \times \Gamma \times \Gamma$ where $\Gamma = 2^{\{\mathsf{OPEN},\mathsf{CLOSED}\}}$.

The *language* of a summary graph is intuitively the set of XML templates that can be obtained by unfolding it, starting from a root node and plugging elements, templates, and strings into gaps according to the edges. A template edge $(n_1, g, n_2) \in T$ informally means that n_2 may be plugged into the g gaps in n_1, and a string edge $S(n) = L$ means that every string in L may be plugged into the gap in n. The gap presence map, which we will not explain in further detail here, is needed during the data-flow analysis to determine where template gaps and attribute gaps occur. (For the curious reader, this is all formalized in [15].) We also define the language of an individual node n in a summary graph: this is simply the language of the modified summary graph where R is set to $\{n\}$.

As an example (borrowed from [15]), we can define a summary graph whose language is the set of `ul` lists with zero or more `li` items that each contain a string from some language L. Assume that the fixed structure is given by $N_\mathcal{E} = \{1,4\}$, $N_\mathcal{A} = \emptyset$, $N_\mathcal{T} = \{2,3,5\}$ (where all three are sequence nodes), $N_\mathcal{C} = \{6\}$, $contents(1) = 2$, $contents(4) = 5$, $attr(1) = attr(4) = \emptyset$, $name(1) = \{\texttt{ul}\}$, $name(4) = \{\texttt{li}\}$, $gaps(2) = \texttt{items}$, $gaps(3) = \texttt{g} \cdot \texttt{items}$, and $gaps(5) = \texttt{text}$. The remaining components are as follows:

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

(For simplicity, we ignore the gap presence map.) This can be illustrated as follows:



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

For a given program, the family of summary graphs forms a finite-height lattice, which is used in the data-flow analysis. To determine the regular string languages used in the string edges, we use a separate program analysis that provides conservative approximations of the possible values of all string expression in the given program [6].

We now introduce two small modifications to the definition of summary graphs:

1. We let the *name* function return a regular set of names, rather than a single name. This will be used to more easily model name classes in Restricted RELAX NG. The definition of unfolding is generalized accordingly: unfolding an element node n yields an element whose name can be any string in $name(n)$, and similarly for attribute nodes. In case an unfolding leads to an element with two attributes of the same name, one of them is chosen arbitrarily and overrides the other.

   To accommodate attribute declarations that have infinite name classes and are repeated using `zeroOrMore` or `oneOrMore`, we define the unfolding of an attribute node n where $name(n)$ is infinite such that it may produce more than one attribute.

2. We distinguish between two kinds of template nodes: *sequence nodes* and *interleave nodes*. The former have the meaning of the old template nodes; the latter will be used to model `interleave` patterns. We define the unfolding of an interleave node as all possible interleavings of the unfoldings of its gaps.

The data-flow transfer functions for operations remain as explained in [15] with only negligible changes as consequence of the modifications of the summary graph definition, the only exceptions being the ones we address in the following section.

Reflecting the [interleaved content] requirement in Restricted RELAX NG, interleave nodes never appear nested within content model descriptions[1]. The translation from Restricted RELAX NG to summary graphs presented in the next section and the transfer functions maintain this property of interleave nodes as an invariant.

With the generalization of the *name* function, we can in fact now easily model computed names of elements and attributes—provided that we add operations for this in the `XML` class, of course, and we leave that to future work.

5 A Translation from Restricted RELAX NG to Summary Graphs

To define the transfer functions for the operations `get` and `cast`, we need an algorithm for translating the given schema type into a summary graph that has the same language. In [15], it is shown how this can be done for DTD schemas; we now present a modified algorithm that supports Restricted RELAX NG and then rely on the translation from XML Schema to Restricted RELAX NG to map from schema types to patterns.

Intuitively, this translation is straightforward: we may simply view summary graphs as a graphical representation of Restricted RELAX NG patterns, provided that we ignore the gap presence component of the summary graphs and the regularity requirement in Restricted RELAX NG. Due to the connection between RELAX NG and regular expression types, this translation can also be seen as a variant of the translation between regular expression types and summary graphs shown in [4].

Given a Restricted RELAX NG pattern, we construct a summary graph fragment as follows:

- First, we observe that name classes and simple-type patterns all define regular string languages[2]. Namespaces are handled by expanding qualified names according to the applicable namespace declarations.

- For an `element` pattern, we exploit the syntactic restrictions described in Section 4. An `element` pattern generally consists of a name class, a content model, and a collection of attribute declarations. Thus, we convert it to an element node e and a template node t with $contents(e) = t$. We define $name(e)$ as the regular string language corresponding to the name class. The attribute declarations are converted recursively into attribute nodes (as explained below), and $attr(e)$ is set accordingly. The content models is converted recursively into a summary graph fragment rooted by t.

- An `attribute` pattern is converted into an attribute node a. We define $name(a)$ in the same way as for `element` patterns, and $S(a)$ is set to the regular string language corresponding to the sub-pattern describing the attribute values. If the attribute is declared as optional using the `optional` pattern, the gap presence map is set to record this (as in [15]).

- For patterns describing content models of elements, the patterns `text`, `group`, `optional`, `zeroOrMore`, `oneOrMore`, `choice`, and `empty` are handled exactly as the equivalent constructs in DTD content model definitions in the way explained in [15]. Intuitively, each pattern corresponds to a tiny summary graph fragment that unfolds to the same language. A `data` pattern becomes a chardata node s where $S(s)$ is the corresponding regular string language. The `interleave` pattern is translated in the same way as `group`, except that an interleave node is used instead of a sequence node.

- Finally, the `notAllowed` pattern can be modeled as a template node t where $gaps(t) = g$ for some gap name g and t has no outgoing template edges.

The set of root nodes R contains the single node that corresponds to the whole pattern being translated. Recursion in pattern definitions simply results in loops in the summary graph. The constructs from RELAX NG that we have omitted in the description in Section 3 can be handled in a similar way as those mentioned here. Note that the translation is exact: the language of the pattern is the same as the language of the resulting summary graph.

As an example, translating the pattern

[1] To state this more precisely, we first define that a node A *top-level-contains* a node B if A and B are identical or B is reachable from A where contents of element nodes and attribute nodes are ignored, and a *content node* is a node that top-level-contains at least one element node or chardata node. We now require the following: every node that has a child that top-level contains an interleave content node must be a sequence or element node, and a sequence node that top-level contains an interleave content node must have only one content node child.

[2] We here ignore a few constraining facets that may be used on the datatypes `float` and `double`. These are uncommon cases that can be accommodated for without losing precision by slightly augmenting the definition of string edges.

```
element ul { element li { xsd:integer }* }
```

results in the summary graph shown in Section 4, assuming that L is the language of strings that match `xsd:integer`.

6 Validating Summary Graphs

When the data-flow analysis has computed a summary graph for each XML expression in the XACT program, we check for each `analyze` operation that the language of its summary graph is included in the language of the specified schema type. If the check fails, appropriate validity warnings are emitted. The entire analysis is sound: if no validity warnings show up, the programmer can be sure that, at runtime, the XML values that appear at the program points marked by `analyze` operations will be valid relative to the given schema types.

The old summary graph analyzer used in XACT is described in [5]. That algorithm, which supports DTD through an embedding into DSD2, as mentioned earlier, has proven successful in practice. We here describe a variant that works with Restricted RELAX NG instead of DSD2.

Given a summary graph node $n \in N_{\mathcal{E}} \cup N_{\mathcal{T}}$ and a Restricted RELAX NG pattern p where p is an `element` pattern, a simple-type pattern, or a complex-type pattern (as defined in Section 3), we wish to determine whether the language of n is included in the language of p.

We begin by considering the case where n is not an interleave node and p is not an `interleave` pattern. First, a context-free grammar C is constructed from the part of the summary graph that is top-level contained by n, considering element and chardata nodes as terminals, template nodes as nonterminals, and ignoring attribute nodes. Each chardata node terminal c is then replaced by a regular grammar equivalent to $S(c)$. If C is not linear, we apply a regular over-approximation [16] (which we also use in [6]). Thus, we have a regular string language L_n over element nodes and Unicode characters that describes the possible unfoldings of n (ignoring attributes). Similarly, p defines a regular string language L_p over `element` patterns and Unicode characters. To obtain a common vocabulary, we now replace each element node n' in L_n by $\langle name(n') \rangle$ (where $\langle$ and $\rangle$ are some otherwise unused characters), and similarly for the `element` patterns in L_p. Then, we check that L_n is included in L_p with standard techniques for regular string languages. (This works because of the restriction to single-type grammars.) If this check fails, a suitable validity error message is generated. Otherwise, for each pair (n', p') of an element node in L_n and an `element` pattern in L_p where $name(n')$ and $name(p')$ are non-disjoint, we perform two checks. First, we check recursively that the language of $contents(n')$ is included in the language of the content model of p'. Second, we check that the attributes of n' match those of p': for each attribute node $a \in attr(n')$, each name $x \in name(a)$, and each value $y \in S(a)$, a corresponding `attribute` pattern must occur in p'—that is, one where x is in the language of its name class and y is in the language of its sub-pattern; also, `attribute` patterns occurring in p' that are not enclosed by `optional` patterns must correspond to one of the non-optional attribute nodes. Again, a suitable validity error message is generated if the check fails.

For interleave nodes and `interleave` patterns, we exploit the restriction on these constructs: they cannot appear nested within content model descriptions. Additionally, in RELAX NG, the sub-patterns of an `interleave` pattern must be disjoint (that is, no element name or `text` pattern occurs in more than one sub-pattern). Thus, if p is an `interleave` pattern, we simply test each sub-pattern in turn, projecting L_n onto the element names occurring in the sub-pattern, and then check that all element names occurring in L_n also occur in one of the sub-patterns. If n is an interleave node, we use a generalized product construction to check inclusion (specifically, the `shuffleSubsetOf` operation in [21]).

To avoid redundant computations (and to ensure termination, in case of loops in the summary graph or recursive definitions in the schema) we apply memoization such that a given pair (n, p) is only processed once. If a loop is detected, we can coinductively assume that the inclusion holds.

With this algorithm, we check for each root node $n \in R$ that its language is included in the language of the pattern corresponding to the given schema type.

As an example of the case with an element node and an `element` pattern, let n be element node 1 in the summary graph from Section 4 and let p be the pattern shown in Section 5:

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

The context-free grammar for the contents of L_n has the following productions (where N_2 is the start nonterminal and N_4 is the only terminal):

$$N_2 \rightarrow N_2^{items}$$
$$N_2^{items} \rightarrow N_3 \mid \varepsilon$$
$$N_3 \rightarrow N_3^{g}\, N_3^{items}$$
$$N_3^{g} \rightarrow N_4$$
$$N_3^{items} \rightarrow N_3 \mid \varepsilon$$

This grammar is linear, so the regular approximation is not applied. The pattern p contains a single sub-pattern

$p' =$ `element li { xsd:integer }`

and by recursively comparing node 4 and p' we find out that the language of node 4 is included in the language of p'. We now see that $L_n \subseteq L_p$, so we conclude that the language of element node 1 is in fact included in the language of the pattern.

With the exception of the regular approximation of the context-free grammars mentioned above, the inclusion check is exact. Also, since the schemas already define only regular languages, the approximation can only cause a loss of precision if the XML transformation defined by the XACT program introduces non-regularity in the summary graphs, and our experience from [15] and [5] indicate that this rarely results in false errors. In particular, the trivial identity function, which inputs XML data using `get` with some schema type and immediately after applies `analyze` with the same schema type, is guaranteed to type check without warnings for any schema type. Moreover, we could replace the approximation by an algorithm that checks inclusion of a context-free language in a regular language, if full precision is considered more important than performance.

An obvious alternative approach to the algorithm explained above would be to exploit the connection with regular expression types and apply the results from the XDuce project for checking subtyping between general regular expression types [10] or to build on Antimirov's algorithm as in the XOBE project [13]. Our main argument for choosing the algorithm explained above is that it has been shown earlier that this approach is efficient for XACT programs. Also, unlike [23], our algorithm behaves much like existing XML Schema validators, but validating summary graphs instead of individual XML documents. Still, the relation between these different inclusion checking algorithms is worth a further investigation.

As an interesting side-effect of our approach, we get an inclusion checker for Restricted RELAX NG and hence also for XML Schema and DTD: given two schemas, S_1 and S_2, convert S_1 to a summary graph SG using the algorithm described in Section 5 and

then apply the algorithm presented above on SG and S_2. (Alternatively, the algorithm presented above could be modified to work directly with Restricted RELAX NG schemas instead of summary graphs.) Preliminary results indicate that our approach is efficient: on a standard PC, our implementation finds in a few seconds the elements in XHTML 1.0 Transitional that are invalid according to XHTML 1.0 Strict (and conversely, it reveals that Strict does not imply Transitional, to our surprise). For schemas that go beyond local tree grammars and use type derivations and `all` model groups, we observe a similarly acceptable performance. Moreover, the validator provides precise error messages in case validation fails.

As an interesting bonus feature, our validator can trivially be extended to precisely check element prohibitions (for example, that `form` elements must not contain `form` elements in XHTML): in XACT, we already have a technique for evaluating XPath location paths on summary graphs, and element prohibitions can be expressed as (simple) XPath location paths.

7 Optional Type Annotations

We will now extend XACT with optional type annotations such that programmers may declare the intended schema types for XML template variables, method parameters, and return values. Besides being useful as in-lined documentation of programmer intentions, type annotations can lead to better modularity properties of the validity analysis.

Every `XML` type may now optionally be annotated in the following way where S and $T_1,\dots,T_n$ are schema types and $g_1,\dots,g_n$ are gap names:

XML<S[$T_1\ \ g_1,\dots,T_n\ \ g_n$]>

The semantics of an annotated type is the language described by S under the assumption that every occurrence of gap g_i has been plugged with a value in the language of schema type T_i.

In XML template constants, every template gap must now have the form <[T g]>, where T is a schema type and g is the gap name. This allows us to, at runtime, tag each gap g in an XML template with a schema type.

In gap annotations in `XML` declarations and template constants, we permit Kleene star of a schema type, T`*`, meaning that the gap can be filled with a sequence of values from the language of T. Kleene star annotations are occasionally needed because we cannot always find existing schema types for sequences of values. As an example, the XML Schema description of XHTML has no named content type describing a sequence of `li` elements. Theoretically, we could permit type annotations to be arbitrary regular expressions over schema types or even small inlined XML Schema fragments, but we have not yet observed the need for this.

Every assignment of an XML template v to a variable x whose type annotation is $t = S$[$T_1\ \ g_1,\dots,T_n\ \ g_n$] must, at runtime, satisfy three constraints:

- All gaps occurring in v must be declared in t.
- For every gap g occurring in v, the language of its type tag must be included in the language of the schema type for g as declared in t.
- The value v must, under the assumption that all gaps were plugged according to their type tags, belong to the language of S.

We put similar constraints on return statements and method invocations, except that for return statements the return value is compared with the declared return type, and for method invocations every actual parameter value is compared with the corresponding declared parameter type. Moreover, every `plug` operation must respect gap tags, that is, the value being plugged in to a gap g must belong to the language of the tag of g.

The following describes a modification of our existing static program analysis to support checking of the extra constraints introduced by annotations.

First, the abstract representation of sets of XML templates is extended to also keep track of the declared schema types of gap names. For a given XACT program, we let $\mathcal{T}$ denote the finite set of all types mentioned by gap annotations in template constants, and we introduce a new summary graph component $D : G \to \mathcal{T}$ mapping gap names to their declared type. The language of a summary graph is not affected by this change.

This leads to extending the data-flow transfer function for the `constant` operation to generate a summary graph with mappings $D(g) = T$ for every gap <T g> occurring in the given XML template constant. (A simple syntactical check ensures that in each template constant all gaps of the same name are declared with identical schema types.) The transfer function for the `plug` operation simply unions the D mappings of its arguments. (Conflicts are avoided by a check mentioned below.) All other transfer functions act as the identity on the new D component.

To ensure type consistency of variables declared with annotated `XML` types, we must validate all assignments to such variables. We check, using the validation algorithm described in Section 6, that the language of the inferred summary graph for the right-hand side of an assignment is a subset of the language permitted by the schema type annotation. However, this inclusion check is modified to treat gaps as if they were plugged with values corresponding to their declared types. More precisely, for every gap g in the inferred summary graph we apply the algorithm described in Section 5 to construct a summary graph fragment SG_g corresponding to the schema type $D(g)$ and then add template edges from all occurrences of g to the roots of SG_g.

To ensure type consistency of template gaps, we perform an additional check of every x`.plug(`g`,`y`)` operation using the summary graphs SG_x and SG_y inferred by the data-flow analysis for x and y, respectively. First, we check that the language of SG_y is a subset of the language of $D_x(g)$ declared for g in SG_x using the inclusion algorithm presented in Section 6. Then, we check that all gap names h occurring in both SG_x and SG_y are declared with identical types, that is, $D_x(h) = D_y(h)$.

As a product of the guaranteed type consistency of variables declared with annotated `XML` types, reading from a variable can now use the declared type instead of the inferred one. More precisely, for every read from an `XML` typed variable x we normally use an inferred summary graph to describe the set of possible template values at that program point, but now, since all assignments to x have already been checked for validity with respect to the declared schema type for x, we can instead apply the algorithm from Section 5 to obtain the summary graph corresponding to the declared schema type.

Note that the support for type annotations leads to a programming style where the explicit `analyze` operation is rarely needed—instead, one may request a static type check by assigning to an annotated variable. This is the style required in other XML transformation languages.

It is well-known that type annotations in programming languages enable more modular type checking. A component, whose interface is fully annotated, can be type checked independently of its context, and type checking the context can be performed without considering the body of the component. In our setting, this, for example, corresponds to methods where all `XML` typed parameters and return

types are annotated, and further, every non-local assignment and read within the method body involves fields declared with annotated types (the latter to constrain side-effects through field variables). As discussed in Section 1, annotations also have drawbacks, however, in XACT, type annotations are optional. This allows the programmer to mix annotated and unannotated `XML` types to get the best from both worlds.

8 Conclusion

We have presented an approach for generalizing the XACT system to support XML Schema as type formalism and permit optional type annotations. Compared with other programming languages for type-safe XML transformations, type annotations are permitted but not mandatory, which allows the programmer to balance between the pros and cons of type annotations.

The extension to XML Schema takes advantage of connections between XML Schema, RELAX NG, and summary graphs. In particular, it involves a tractable subset of RELAX NG that we use as an intermediate language in the static analysis.

The ideas presented in this paper will become available in the next version of the XACT implementation.

References

[1] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *Proc. 19th European Conference on Object-Oriented Programming, ECOOP '05*, volume 3586 of *LNCS*. Springer-Verlag, July 2005.

[2] Paul V. Biron and Ashok Malhotra. XML Schema part 2: Datatypes second edition, October 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-2/`.

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

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

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

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

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

[8] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. The Xtatic experience. Technical Report MS-CIS-04-24, University of Pennsylvania, October 2004. Presented at Programming Language Technologies for XML, PLAN-X '05.

[9] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML processing in Java. In *Proc. 14th International Conference on World Wide Web, WWW '05*, pages 278–287. ACM, May 2005.

[10] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

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

[12] Kohsuke Kawaguchi. Sun RELAX NG Converter, April 2003. `http://www.sun.com/software/xml/developers/relaxngconverter/`.

[13] 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.

[14] Christian Kirkegaard, Aske Simon Christensen, and Anders Møller. A runtime system for XML transformations in Java. In *Proc. Second International XML Database Symposium, XSym '04*, volume 3186 of *LNCS*. Springer-Verlag, August 2004.

[15] 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.

[16] Mehryar Mohri and Mark-Jan Nederhof. *Robustness in Language and Speech Technology*, chapter 9: Regular Approximation of Context-Free Grammars through Transformation. Kluwer Academic Publishers, 2001.

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

[18] Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, 2005.

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

[20] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML schema languages using formal language theory. In *Proc. Extreme Markup Languages*, August 2001.

[21] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2005. `http://www.brics.dk/automaton/`.

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

[23] Akihiko Tozawa and Masami Hagiya. XML Schema containment checking based on semi-implicit techniques. In *Proc. 8th International Conference on Implementation and Application of Automata, CIAA '03*, volume 2759 of *LNCS*, July 2003.