Institutt for informatikk

Towards integration of XML in the Creol object-oriented language

Arild Torjusen, Olaf Owe, and Gerardo Schneider

Research report 365
ISBN 82-7368-323-0

October 2007

Towards integration of XML in the Creol object-oriented language

Arild Torjusen, Olaf Owe, and Gerardo Schneider

Department of Informatics, University of Oslo
PO Box 1080 Blindern, NO-0316 Oslo, Norway
email: {aribraat,olaf,gerardo}@ifi.uio.no

Abstract

The integration of XML documents in object-oriented programming languages is becoming paramount with the advent of the use of Internet in new applications like web services. Such an integration is not easy in general and demands a careful language design. In this paper we propose an extension to Creol, a high level object-oriented modeling language for distributed systems, for handling XML documents.

1 Introduction

XML (eXtensible Markup Language) [**?**] is a flexible and generic format for structured data aimed at being shared on the World Wide Web and intranets. The need of XML documents as first-class citizens has been identified few years ago both at the academic as well as by business-oriented communities [**?**]. XML *documents* are ordered labeled tree structures containing *markup* symbols describing their content. The document structure is described by a document type -or *schema*- written in a schema language. Many such languages have been proposed, among them DTD (Document Type Definition) [?] and XML-Schema [**?**]. Unlike other markup languages (like HTML), XML has no restrictions on the tags or attributes used to mark up a document. One remarkable feature of XML is its plain-text-based nature. The advantage is that there is no problem with proprietary nor deciphering data. The disadvantages are the large bandwidth needed for transmission of documents and the need of encryption because of security issues. Part of the manipulation of XML documents includes the retrieval of information through queries. XQuery [**?**] provides a sound foundation for XML query, based on infosets. The situation is not ideal for developers since they need to know one language for analyzing the tuples e.g., SQL, another language for the Infoset e.g., XQuery, and a third one for operating on objects e.g., Java. Some attempts have been done to combine object-oriented languages and XML, but this turned out to be a complex task; this problem is known as the *impedance mismatch* [**?**], which arises when trying to combine object-oriented programming languages and (relational) databases.

The integration of XML on current object-oriented languages is far from trivial. The initial approach has been to treat XML through APIs which uses strings for representing literals. One problem of this approach is that it limits the use of static checking tools. Furthermore, the representation of programs as text involves potential security risks. See [?] for a more detailed description of the main problems arising with the integration of XML in object-oriented languages.

In addition to the integration of XML documents within OOP languages, another question is what to do with these data, i.e., how easy it is to make queries, getting useful information from such XML-documents.

1.1 Creol

In this paper we are concerned with a light-weight integration of XML into the object-oriented language Creol [?, ?, ?]. Our motivation for choosing Creol may be summarized as follows:

- It supports both object-oriented classes, with late binding and multiple inheritance, as well as user defined data types and functions. This gives flexibility in our choices when representing XML.
- It is oriented towards open distributed systems. Exchange of XML documents fits naturally in this context.
- It supports concurrency and method calls based on asynchronous communication. We wish to explore the processing and sharing of XML documents in this setting.
- It is strongly typed, supporting subtypes and subinterfaces, with a type hierarchy including both by means of the universal type, *Data*.
- It has a formal operational semantics, defined in rewriting logics. This enables us to formalize the extension to XML by reuse of the operational semantics.
- It has a small kernel with an operational semantics consisting of only 11 rewrite rules. This makes it easy to extend and modify the language and the semantics.
- Creol has an executable interpreter defined in the Maude language. This provides a useful framework for implementation and testing of our XML representations.

1.2 Related Work

The list of languages for processing XML documents is extensive, so it is not possible to be exhaustive here. We briefly discuss below some of the most influential works, namely XDuce, CDuce and C_ω. We mention other related work as reference for further reading, without entering into detail.

XDuce XDuce [?] is a functional programming language for XML processing. Its basic data values are XML documents and its types—called *regular expressions types*—correspond to document schemas. The language is statically typed but it also provides dynamic type-checking. Other interesting feature of XDuce is regular expression pattern matching which includes tag checking, subtree extraction and conditional branching.

An XML document in XDuce is represented as a sequence of nodes, and types use similar constructs as string regular expressions like "*" for representing that zero or more occurrences may happen, "?" for indicating an item may be omitted, "+" for one or more time repetition, "|" for alternation and "," for concatenation. The main difference with string regular expressions, is that regular expression types describe sequences of tree nodes instead of sequences of characters.

The type-checking algorithm is based on the following subtype relationship: one type is a subtype of another if and only if the former denotes a subset of the latter. The subtype checker may be used both for checking that the actual type of a function's body is a subtype of the programmer-declared result type and for verifying function call arguments against parameter types given by the programmer. Although the theoretical complexity of the corresponding problem to subtype checking on tree automata is exponential, it is claimed in [?] that it works well in practice.

CDuce CDuce [**?**] is a typed functional language born from an attempt to solve some of the limitations of XDuce [?]. It extends XDuce on three areas:

Type system In addition to regular expression types and type-based patterns, CDuce adds recursive types and other XML specific constructs: products, records (open and closed), general Boolean connectives (intersection, union and difference) and arrow types. This extension takes care of not breaking down the nice subtype relation of XDuce.

Language design The following language constructions are included in CDuce: overloaded functions (useful for code sharing and reuse), iterators on sequences and trees and other extensions of the pattern algebra. Besides, XML tags are first-class citizens and strings are simple sequence of characters. The language support higher-order programming, so all functions are first-class citizens.

Run-time system A new approach for avoiding unnecessary computation at runtime is added in CDuce, allowing the programmer to use a more declarative style when writing patterns, without degrading performance. The underlying theory is based on a new kind of tree automata.

CDuce provides also a tool for translating DTDs into CDuce's types.

C$_\omega$ C$_\omega$ [**?**] is a programming language developed at Microsoft Research, combining features from two other research languages: (a) Polyphonic C#: a control flow extension with asynchronous wide-area concurrency, and (b) Xen [?]: a data type extension for processing XML and table manipulation. Besides other interesting features, C$_\omega$ allows the construction of objects using XML syntax.

The C$_\omega$ type systems combines the following three data models: relational, object and XML data-access, and it is more oriented to XML constrained using W3C XML Schema. The language covers the following XML and XML Schema features: document order, distinction between elements and attributes, multiplicity of fields with equal name but different values and content models for specifying choice (union) types for fields.

One of the nice features of the C$_\omega$ type system are *streams*. It is possible to invoke methods on streams, which are applied to all the elements of the stream; XPath-style queries over objects graphs are easily written in this way. It also includes the concept of *apply-to-all* expressions construct. Choice (union) types allow the programmer to specify one of different possible values for a certain field. Moreover, *null* is a valid value for a type, which have been proved useful in XML and relational databases. Document order and multiplicity of equal names for child elements, are solved through the use of *anonymous structs*. In C$_\omega$ DTDs (and XML Schemas) are represented by *content classes*.

Other languages The following languages try to extend Java with XML processing: XJ [**?**], XACT [**?**], XOBE [**?**], BPELJ [**?**].

XL [**?**] is a language whose only type system is the XML type system, and not a language whose syntax is described using XML vocabulary. It is specially designed for the implementation of Web services. XL is portable and fully compliant with all W3C standards such as XQuery, XML Protocol, and XML Schema.

PiDuce[1] is CDuce-like language based on the π-calculus. ECMAScript for XML (E4X) is a set of programming language extensions adding native XML support to ECMAScript. E4X is standardized by Ecma International in ECMA-357 standard.[2]

See [**?**] for a good survey on static type-checking for XML transformation languages.

1.3 Our Agenda

In order to integrate XML documents in Creol, we intend to follow the following agenda:

1. *Parsing and well-formedness checking.* We will enhance the language as to be able to take a given XML document as input and generate some internal data structure from it.

2. *Internal representation of XML in Creol.* We aim at extending Creol for supporting XML documents with the least possible changes to the existing framework. One of the key features we would like to preserve is Creol static type-safety. In order to make a lightweight integration of XML into Creol and keep static type safety we will restrict type checking of XML in this implementation to only *well-formedness* of XML values, i.e. that some value of type `XMLDoc` (the Creol type for XML documents) checks out as an `XMLDoc`.

3. *Simple validity-checking of XML data-structures.* We will validate XML data-structures against some schema. Schema is here taken in a broad sense, meaning a formal description of the type of an XML document, without regards to any specific schema language as e.g. DTD, XML-Schema or RELAX NG (cf. Sec 3). Validity checking will be done by functions "on top" of the type system and not within the type system itself.

4. *More complex validity-checking of XML data-structures.* We will perform more complex validity checking after enhancing the Creol language with *regular expression types*, following the work of Hosoya et.al. [**?**].

5. *Queries.* We will also demonstrate how to perform queries and data extraction from XML document instances.[3]

6. *Transformations.* We will perform more complex operations such as construction and transformations on XML documents.

In this paper, however, we will concentrate on items 2 and 3 above. In the next section we show how XML documents are integrated in Creol. In Section 3 we show how schemas are represented in Creol after a short discussion on existing schema languages. Section 4 is concerned with the validation of XML documents. In Section 5 we conclude and present further work.

[1]`http://www.cs.unibo.it/~laneve/PiDuce/`

[2]See `http://www.ecma-international.org/publications/standards/Ecma-357.htm`.

[3]Cf. e.g. `http://www.w3.org/TR/2005/WD-xquery-use-cases-20050915/` for test use cases.

2 A model for XML in Creol

Different XML documents may vary in physical representation due to syntactic changes permitted by the XML standard. W3C has issued a recommendation which describes how any XML document can be normalized into a canonical form [?]. The data model defined in the XPath 1.0 Recommendation [?] is the basis for canonical XML and we will use this as the point of departure for the internal representation of XML in Creol.

2.1 The XPath Data model

XPath models an XML document as an ordered tree containing nodes of seven different types:

- root: The root node is the root of the tree and will correspond to an XML document instance. It contains a list of processing instructions, a list of comment nodes, and exactly one element which is the root element of the document.
- element: The element node has a name (corresponding to the xml tag for the element) and may have as its children element nodes, comment nodes, processing instruction (PI) nodes and text nodes. It is also associated to a set of attribute nodes and a set of namespace nodes.
- text: A text node contains a string, representing character data in the XML document.
- attribute: An attribute node contains a name and a value.
- namespace: A namespace node contains a string value for the namespace prefix and a value for the namespace URI.
- processing instructions: A PI node has a name identifying the target application and a string which is to be passed to the application.
- comment: A comment node contains a string.

To simplify the initial XML implementation for Creol we will leave out the last three kinds of nodes from our model. According to [?], comments "are not part of the document's character data; an XML processor MAY, but need not, make it possible for an application to retrieve the text of comments.", we choose not to retain comments in the Creol representation of XML. Processing instructions are not relevant for our purpose of demonstrating lightweight integration of XML in Creol and can also be left out. As will be explained later we will adopt the DTD-language for specification of schemas; since the DTD does not support namespaces it is natural not to represent namespace nodes in the model. These design choices also simplifies the definition of element and root nodes.

2.2 The Creol representation of XML

Given the two-tiered type-system of Creol where objects are typed by interfaces and local computations on terms occur in a functional language, we introduce XML into Creol by adding type constructors for a new `XMLDoc` type, as a subtype of the universal type `Data`, as well as functions on this type.

Creol has an operational semantics defined in rewriting logic, which is executable with Maude [?] and provides an interpreter and analysis platform for system models. So to accommodate XML we extend the operational semantics with some Maude sorts (type names) and constructors (Creol definitions would be very similar):

```
sorts   XMLName ElemNd TextNd AttNd ContentNd XMLDoc .
subsort ElemNd TextNd < ContentNd .
```

introducing sorts for XML names, element, attribute, text and content nodes, letting `ElemNode` and `TextNd` be subsorts of `ContentNode`.

To simplify the writing of XML values in a program we use mix-fix notation (indicating argument positions by underline) to provide a compact syntax by adding the following constructors for attributes, textnodes and elements (with and without attributes).

```
op (_=_)    : XMLName String                   -> AttNd  [ctor] .
op _(_)[_]  : XMLName AttNdList ContentNdList -> ElemNd [ctor] .
op _[_]     : XMLName ContentNdList           -> ElemNd [ctor] .
op tx       : String                           -> TextNd [ctor] .
```

where the clause [ctor] after an operator (op) indicates that it is a constructor.

Note that there is no specific constructor for root nodes. Since we leave out processing instructions and comments, the root node is just the element node occurring at the root of an XML document tree. Thus, the XML document constructor is:

```
op xmlDoc   : ElemNd XMLSchema -> XMLDoc [ctor] .
```

We define the operator

```
op noSchema :                   -> XMLSchema [ctor] .
```

for XML documents with no XMLSchema. Other XMLSchema constructors are defined further below.

Example The following simple XML fragment,

```
<email>
  <head>
    <sender>Arild</sender>
    <recipient mailaddr="vera@foo.com">Vera</recipient>
    <subject>Test</subject></head>
  <body>
    <message>Hello there, you wrote in an earlier message:
    <quote>We'll meet again</quote> See you later</message>
  </body>
</email>
```

has the Creol/Maude syntax:

```
"email"[
  ("head"[
    ("sender"[tx("Arild")])
    ("recipient"("mailaddr"="vera@foo.com")[tx("Vera")])
    ("subject"[tx("Test")])])
  ("body"[
    ("message"[
    tx("Hello there, you wrote in an earlier message:")
    ("quote"[tx("We'll meet again")]) tx("See you later")])])] .
```

As conventional in Maude, the list constructor (concatenation) is here denoted by white space (blank).

3 Schemas and type checking

3.1 Schemas as types vs. schemas as values

Static type checking of XML documents in a programming language can be achieved by introducing a type for XML schemas in the language. Xduce and CDuce mentioned earlier are examples of projects going in this direction.

For the current integration of XML in Creol we will take a less involved approach by introducing a data type for schemas, together with functions to validate a document against a schema. This takes place within the existing type system and does not constitute an addition to the type system itself.

3.2 Expressive power of schema languages

There exists several generally adopted XML schema languages with different expressive power. Murata, Lee, and Mani [**?**] suggest a taxonomy of schema languages based on the formal theory of regular tree grammars. Some of the most common schema languages can be ranked in order of increasing expressivity thus: The DTD-language, The W3Cs XML Schema, The RELAX NG specification. Validation of the first two can be done by simple adaptions of word automata, while the last requires a more complicated tree automaton. However the DTD language is sufficiently expressive for our purpose which is to demonstrate how XML can be integrated in the object oriented modeling framework of Creol. Therefore in our model for XML schema values in Creol we adapt the restrictions inherent in the DTD language to achieve simple validation, (i.e. only deterministic regular expressions is allowed in the definition of an element as explained below).[4]

3.3 The schema type for Creol

A DTD is a list of markup declarations where markup declarations are either element type declarations, attribute-list declarations, entity declarations, or notation declarations.

For our purpose we only consider element type declarations and attribute-list declarations. Entity declarations may be considered as a kind of macro notation for strings that may appear in a DTD or an XML document, since our focus is on internal processing we will assume that these already are expanded by the parser and will abstract away from them in our model. Notation declarations are similarly a kind of shorthand for notations and are also left out. Accordingly the `XMLSchema` constructor is:

```
op xmlSchema : XMLName ElemDeclList AttDeclList -> XMLSchema .
```

Element type declarations consist of a name referring to an element and a specification of the legal content. There are four kinds of specifications: either one of the designated keywords “EMPTY” or “ANY”, or the specification of a *content model*. A content model is a context free grammar governing the allowed types of the child elements and the order in which they are allowed to appear. The fourth kind of content specification is the *Mixed-content Declaration* which is of the form:

$$(\,\texttt{\#PCDATA}\,|\,e_1\,|\,e_2\,|\,\ldots\,|\,e_n\,)\,*$$

Where each e_i is an element name and n may be 0 in which case the ’*’ is optional.

[4] Roughly corresponding to “Local Tree Grammars” in [?].

Example A DTD for the XML fragment given above could be:

```
<!DOCTYPE email [ <!ELEMENT email (head, body, foot*) >
<!ELEMENT head (sender, recipient, subject?)>
<!ELEMENT body (message)*>
<!ELEMENT foot (#PCDATA)><!ELEMENT sender (#PCDATA)>
<!ELEMENT recipient (#PCDATA)><!ELEMENT subject (#PCDATA)>
<!ELEMENT message (#PCDATA|quote)*><!ELEMENT quote (#PCDATA)>]>
```

The first three element declarations specify content models and the rest are instances of mixed-content declarations. We model the content models as *regular expressions*. Let Σ be an alphabet over element names, including the reserved name PCDATA. By including PCDATA in Σ we can model a mixed-content declaration as a special kind of a content model specification. The set of regular expressions over Σ^* are obtained in the standard way: The empty string ϵ and each member of Σ are regular expressions. If α is a regular expression, then so are (α), $\alpha?$, $\alpha*$ and $\alpha+$. If α and β are regular expressions, then so is $\alpha\ \beta$, and $\alpha\,|\,\beta$. The operators ?, $*$, and $+$ has higher precedence than concatenation. Concatenation has higher precedence than union ($|$). The regular expression combinators have the expected semantics. We model element declarations as follows:

```
subsort XMLName < RegExp .
op elDecl      : XMLName ContentModel   -> ElemDecl [ctor] .
ops empty any  :                        -> ContentModel  [ctor]  .
op elCt        : RegExp                 -> ContentModel [ctor] .
op PCDATA      :                        -> RegExp  .
ops _? _* _+   : RegExp                 -> RegExp [ctor prec 40 ] .
op (_@_)       : RegExp RegExp          -> RegExp [ctor assoc prec 42 ][5]
op _|_         : RegExp RegExp          -> RegExp [ctor prec 44 ]
```

The XML specification adds the requirement that the content models must be deterministic [?, Appendix E], i.e. a content model must not allow an element to match more than one occurrence of an element name in the content model. This ensures that when matching an element name σ with a schema we do not have to look ahead beyond the σ in the input string to decide which regular expression in the content model matches σ. The requirement is included in the XML specification to ensure compatibility with SGML. For a detailed discussion see e.g. [**?**].

Example The maude syntax for the DTD given above is:

```
xmlSchema("email",(
elemDecl("email",elCt("head"@"body"@("foot"*)))
elemDecl("head",elCt("sender"@"recipient"@("subject"?)))
elemDecl("body" , elCt("message"*))
elemDecl("foot", elCt(PCDATA)) elemDecl("sender" ,elCt(PCDATA))
elemDecl("recipient" ,elCt(PCDATA)) elemDecl("subject" ,elCt(PCDATA))
elemDecl("message" ,elCt((PCDATA|"quote")*))
elemDecl("quote" ,elCt(PCDATA)),noAttDecl[6]) .
```

4 Validating XML in Creol

Well-formedness of any value of type `XMLDoc` is ensured by Maude type checking. The XML specification defines an XML document to be *valid* "if it has an associated document type declaration and if the document complies with the constraints expressed in it" [?].

[5]We use '`@`' as the concatenation operator to avoid problems with overloading of ',' or whitespace which might otherwise have been used.

[6]Attribute declarations are not yet supported.

```
op res : Bool String -> ValResult .
eq collate( res(b,s) , res(b',s')) = res((b and b') , (s + s')) .
eq validate( xmlDoc(elemNd(nm,atts,cts) , noSchema )) = res(false,"No Schema") .
eq validate( xmlDoc(elemNd(nm,atts,cts) , xmlSchema(nm',elDs,attDs) ) ) =
  if ( nm =/= nm' ) then
    res(false,("Document root-element: " + nm +
               ", must match schema type: " + nm' + "\n"))
  else
    val(elemNd(nm,atts,cts) , elDs)
  fi .
eq val(emp,elDs) = res(true,"") .
eq val((ct cts), elDs) = collate(val(ct,elDs),val(cts,elDs)) [owise] .
ceq val(elemNd(nm,atts,cts),elDs) = if cm == undefined then
    res(false,("Element-type :" + nm + " must be declared.\n"))
  else
    check(elemNd(nm,atts,cts),cm,elDs) fi if cm :=  getCM(nm,elDs) .
eq check(elemNd(nm,atts,cts),empty,elDs) =
  if (cts == emp) then res(true,"Empty elem: " + nm + "n")
  else res(false,"Elem: " + nm + " declared as EMPTY, but has content.\n") fi .

eq check(elemNd(nm,atts,cts),any,elDs) =
  collate(res(true,"Elem: " + nm + " defined as ANY.\n"),val(cts,elDs)) .

eq check(elemNd(nm,atts,cts), elCt(regexp) ,elDs) =
  if match(getTokens(cts), regexp) then
    collate (res(true, nm + ": (" + ctToS(cts) +
            ") matches [" + reToS(regexp) + "]n") , val(cts,elDs))
  else
    collate (res(false, nm + ": (" + ctToS(cts) +
             ") does NOT match [" + reToS(regexp) + "]n"), val(cts,elDs) ) fi .
```

Figure 1: Maude code for validation of XML documents.

The XML document constructor associates the root element of a document with a schema, (which may also be the special value `noSchema`). Hence, an XML document is validated by first checking for existence of a schema and by checking that the root node element name matches that schema name. Secondly we check that each element node in the tree is valid with respect to the element declarations in the schema.

Validation on a document is performed by the function

```
op validate : XMLDoc -> ValResult .
```

The validate function checks whether there is a schema associated with the document and whether the schema name matches the document root node, if it does, the recursive function `val` is called, otherwise validation stops. A `ValResult` is a pair of a boolean value and a string, where the boolean value indicates whether the document is valid and the string is used to return an error message or a record of the processing of the document. The helper function `collate` builds the final validation result for a document from validation of its parts. The relevant parts of the maude code are given in fig. 1. The function:

```
op val : ContentNdList ElemDeclList -> ValResult .
```

validates a content node list against the element declaration list defined by the schema. For a list of nodes, `val` is called recursively on each node in the list. For a single node, the element type declaration corresponding to the node is retrieved (by name) from the list of element declarations. If no declaration exists for a content node, the document is invalid, otherwise the node is checked against the retrieved declaration by a call to the function `check`[7]:

```
op check : ContentNd ContentModel ElemDeclList -> ValResult .
```

In the call to `check`, the complete list of element declarations is passed on as a parameter since any child nodes to the node currently being processed must be validated.

For an element to be valid, a declaration for the element must exist and the following should hold: If the content specification is "EMPTY" the element should have no content. If the content specification is "ANY", the element can consist of any sequence of (declared) elements intermixed with character data. If the content specification is a content model, the sequence of child elements must belong to the language generated by the regular expression in the content model. If the content specification is *mixed* the content must consist of character data and child elements whose types match names in the content model [?, Sec. 3].

The function `check` has three cases corresponding to the four validity conditions for elements, one case for each of the specifications EMPTY and ANY and one case for a content model or a mixed specification. The first two cases are easy to check, in the first case we must make sure that the element declared as empty is in fact empty, in the second case no further checking of the element is necessary but we still have to call `val` for each child node of the current node.

For the third case the `check` function will use the function `match` to determine whether the list of actual children elements matches the regular expression specified in the corresponding element declaration, in addition `val` is called for each child node.

The function `getTokens`, builds a list of tokens from the element content, i.e. a list of element names (including the token 'PCDATA). As tokens we use the Maude built-in sort `Qid`. The token list and the regular expression from the element type declaration are processed by the `match` function:

```
op match : TokenList RegExp -> Bool .
```

Matching of a list of element names from Σ against a regular expression is implemented by constructing a deterministic finite automaton from the regular expression and test whether the automaton accepts the string corresponding to the list of names. See e.g. [**?**] for a description of how this is done in Maude. `ctToS` and `reToS` are just string conversion functions for content nodes and regular expressions for logging purposes.

Example Evaluation of the sample document with the DTD specified above gives the following result:

```
res(true, "email: (head ,body) matches [head @ body @ (foot*)]
           head: (sender ,recipient ,subject)
                 matches [sender @ recipient @ (subject?)]
           sender: (PCDATA) matches [PCDATA]
           recipient: (PCDATA) matches [PCDATA]
           subject: (PCDATA) matches [PCDATA]
           body: (message) matches [(message*)]
           message: (PCDATA , quote ,PCDATA)
           matches [(PCDATA | quote*)]
           quote: (PCDATA) matches [PCDATA]") .
```

[7]Note that according to [?] an element type must not be declared more than once so uniqueness of element declarations may be assumed.

5 Conclusion

Integrating XML documents in object-oriented languages is not easy in general as witnessed by the extensive research conducted in this area, and nicely presented in the survey [?]. We have shown here how to integrate XML documents into Creol, an object-oriented language with formal semantics in rewriting logic. We have also presented an algorithm for validating XML documents against XML schemas, to show that the former are instances of the latter.

This paper is a first step towards a full integration of XML into Creol, and we intend to pursue our work as to complete our agenda described in Section 1.3. In particular, we find it extremely interesting to be able to manipulate and reason about XML documents, to include regular expression types, and to adapt the semantic sub-typing algorithm from CDuce and XDuce discussed in the introduction.

References