IST BASIC RESEARCH PROJECT
SHARED COST RTD PROJECT
THEME: FET DISAPPEARING COMPUTER
COMMISSION OF THE EUROPEAN COMMUNITIES
DIRECTORATE GENERAL INFSO
PROJECT OFFICER: JAKUB WEJCHERT

Global Smart Spaces

D16.0 Initial Common Language Implementation

SEPTEMBER 2002, STRATHCLYDE/WP6/V1.0

RICHARD CONNOR, DAVID LIEVENS, PAOLO MANGHI, STEVE NEELY, FABIO SIMEONI AND PADDY NIXON

IST Project Number	IST-2000-26070	**Acronym**	GLOSS
Full title	Global Smart Spaces		
EU Project officer	Jakub Wejchert		

Deliverable	**Number**	D16	**Name**	Initial Common Language Implementation			
Task	**Number**	1	**Name**	Flexible Binding Mechanism			
Work Package	**Number**	WP6	**Name**	Theories of Mobility			
Date of delivery	**Contractual**		October 2002		**Actual**		September 2002
Code name					**Version** 1.0	draft ✍	final ✍
Nature	Prototype ✍ Report ✍ Specification ✍ Tool ✍ Other:						
Distribution Type	Public ✍ Restricted ✍ to:						
Authors (Partner)	R. Connor, D. Lievens, P. Manghi, S. Neely, F. Simeoni and P. Nixon (Strathclyde)						
Contact Person	D. Lievens						
	Email	david@cis.strath.ac.uk	**Phone**	+44 1415484310	**Fax**	+44 1415525330	

Abstract (for dissemination)

The this report discusses a language-level binding mechanism between a typed programming language and XML. Through it, programs written, for example, in Java can safely bind to XML encoded data while maintaining a high degree of resilience to change.

In particular, any changes to the structure of the data that are irrelevant to the binding component do not invalidate the binding. This is significantly better than the binding mechanisms in legacy middleware such as COM or CORBA, where any change to the structure of the data may require a complete redeployment of all the components cooperating in the distributed application -not only those that are directly affected. Such resilience to change is paramount for global smart applications, where redeployment may be awkward for many components (e.g. controlling software of a smart room).

The mechanism is language-specific, but is explored in the paper in terms of a canonical language that is flexible enough to capture many typed programming languages, including the most commonly used imperative and object oriented ones such as Java, C++ and C. The ability to adapt the mechanism for legacy programming languages is very important for GLOSS, as it is to be expected that large-scale smart applications will draw upon existing tools and technologies and may therefore be more conveniently built in one or another existing programming language.

* This paper has been published in Elsevier's Journal on Information and Software Technology 44(2002) pp. 217-228.

Keywords XML, binding mechanism, programming language(s), SNAQue prototype

1. INTRODUCTION

Values of existing typed programming languages are increasingly generated and manipulated outside the language jurisdiction. Instead, they often occur as fragments of XML documents (cf. [1]).

This may be because the containing documents are *semistructured*, i.e. their structure is too irregular or unstable to be effectively handled by traditional programming languages or DBMSs (cf. [2,3]). It may also occur when the document is more disciplined, but needs to be exchanged across proprietary boundaries in a standard and self-describing format.

As an example, consider the following XML document d, where some irregularities have been intentionally added to the data for sake of illustration.

```
<staff>
   <member code = "123517">
      <name>Richard Connor</name>
      <home>www.cis.strath.ac.uk/~richard</home>
   </member>
   <member code = "123345">
      <name>Steve Neely</name>
      <ext>4565</ext>
      <project>
         <name>SNAQue</name>
      <project/>
   </member>
   <member code = "175417">
      <ext>4566</ext>
      <name>Fabio Simeoni</name>
   </member>
</staff>
```

On a much larger scale, this irregularity would prevent the document from being conveniently managed within the typed framework of conventional technology. While union types and object-oriented features may accommodate some of the irregularity, their abuse would soon degrade the performance of the system and complicate program specification and maintenance.

Consider instead the fragment d' of d shown next:

```
<staff>
   <member code = "123517">
      <name>Richard Connor</name>
   </member>
   <member code = "123345">
      <name>Steve Neely</name>
   <member code = "175417">
      <name>Fabio Simeoni</name>
```

```
   </member>
</staff>
```

For most object-oriented languages, d' may be an XML encoding of an object `staff` of class `Staff`, where

```
class Staff {
      private Member[] member;
      Member[] getMembers() {...}
      void setMembers(Member[] members) {...}
      ...}
```

and `Member` is the class:

```
class Member {
      private String name;
      private int code;
      String getName() {...}
      void setName (String n){...}
      int getCode() {...}
      void setCode (int c){...}
      ...}
```

This simple observation raises the expectation that programming over d' be as simple, safe, and efficient as programming over `staff` with existing programming languages. In particular, we require these good properties to scale, i.e. hold for generalised computations over XML fragments considerably larger than d'. Unfortunately, we believe that none of the current approaches fully satisfies such requirement.

1.1. Background

To date, computations over XML data can be specified in a variety of paradigms, models and languages. Two kinds of approaches, however, appear to prevail: dedicated query languages and bindings to programming languages, typically object-oriented ones.

In query languages such as [4–7], queries have a familiar SQL-like structure, but contain powerful path expressions specified against the tree topology of the data. This gives the languages the flexibility required to compute over data with irregular or partially known structure. It makes them also more succinct than full-fledged programming languages for most operations of data filtering and transformation.

However, query languages are usually not Turing-complete, nor well suited to complex programming tasks over large datasets, possibly involving recursion. Furthermore, they are essentially untyped, except with respect to the tree structure of the data. While this is justified in the general case by the typeless nature of the format, potential regularity in (subsets of) the data could and should be exploited for program verification and optimisation.

Language bindings are instead defined by implementing programming interfaces to one of two possible in-memory representations of the data. In the Document Object Model

interface (cf. [8]), the data is organised and manipulated as a labelled tree. In the Simple API for XML (cf. [9]), the data is a string of characters organised and processed along parsing events.

Beside performance-related differences, both solutions impose an interpretation of the data which generalises their structural relationships (e.g. nodes of a tree), but conveys only indirectly and too concretely their intended meaning (e.g. staff of a university department). When computations explicitly address the structural properties of the data (e.g. adding or removing a node, searching for a string in the data), this interpretation is adequate. In most cases, however, it complicates program specification, making it tedious, error-prone and hard to maintain.

Consider, for example, any computation over the names and codes of the staff members in d. In a pure implementation of the DOM interface for a Java-like language, the code may include something like:

```
int code;
String name=null;
Element staff=d.getDocumentElement();
NodeList members =
            staff.getElementsByTagName("member");
int memberCount = members.getLength();
for (int i=0;i<memberCount;i++) {
 Element member = (Element) members.item(i);
 code = Integer.parseInt(member.getAttribute("code"));
 NodeList children = member.getChildNodes();
 int length = children.getLength();
 for (int j=0;j<length;j++) {
  Node child = children.item(j);
  if (child.getNodeType()==Node.ELEMENT_NODE) {
   String tagName= ((Element) child).getTagName();
   if (tagName.equals("name")) name=
    ((characterData) child.getFirstChild()).getData(); }
    ...do something with name and code ...}}
```

Even for a simple task, the code is highly convoluted and inefficient. Partly, this is due to the document-oriented nature of any XML programming interface. For instance, semantically related data must be accessed with the different algebras of elements and attributes. Similarly, manipulating atomic data requires an implicit or explicit cast from the type of strings, the only available. More generally, the logic of the computation is unnecessarily expressed in an algebra of trees, while domain-specific concepts (e.g. names and codes) are relegated to the role of run-time parameters.

The same task could have been specified directly against the object `staff` of class `Staff` defined above, and as simply as:

```
Member[] members = staff.getMembers();
for (int i=0;i<members.length;i++) {
```

```
int code = members[i].getCode();
String name = members[i].getName();
...do something with name and code ...}
```

The code is now aligned to the semantics of the application. It is also more succinct and less redundant, for generic operations on staff and staff members do not have to be repeated within specific computations, but can be factored out in class declarations, thoroughly tested, and then reused.

Inadequate data abstractions also compromise static checking of computations. Correctness can be guaranteed for operations on trees and strings, but not staff members. For example, the following invocation:

```
NodeList members = staff.getElementsByTagName("mebmer");
```

where `"mebmer"` is a typo for `"member"`, would silently compile and return a `null` value only at run-time. Safety is thus responsibility of the programmer, not the system. Programmatic checks worsen readability and maintainability of the code, and are not always sufficient to guarantee correct behaviour. In the lack of some description of the data (e.g. a DTD), the typo may be interpreted as the absence of required data and thus trigger unintended behaviour. Even assuming some data description, the typo may accidentally identify some other data or, in the best case, be simply signalled at run-time.

For similar reasons, the system can optimise resources only within the limits of its static knowledge of the data. For instance, it ignores the fact that all staff members have names and codes.

1.2. Extraction Mechanisms

Motivated by the previous observations, we aim at defining *high-level bindings* between XML and existing programming languages, which preserve the intended semantics of the data.

Specifically, we propose language-specific mechanisms that *extract* self-describing representations of language values from arbitrary XML documents, and transform them into their counterparts within the language. Thereafter, the extracted data are computed over in a familiar, expressive, and robust environment.

To achieve this for a given language, we interpret the extraction of a value as the projection of its language type over the containing XML document. Following the previous example, the projection of class `Staff` over d would result in the extraction of the object `staff`.

More formally, let $\mathbf{D}$ be the set of XML documents, $\mathbf{L}$ a typed programming language, and $\mathbf{V}$ and $\mathbf{T}$ the value and type spaces of $\mathbf{L}$, respectively (see Figure 1). Let also $sd : \mathbf{V} \to \mathbf{D}$ be a self-describing interpretation of $\mathbf{L}$'s values in $\mathbf{D}$, and $\preceq$ in $\mathbf{D} \times \mathbf{D}$ a relation of 'inclusion' between XML documents.

Definition 1.1 *Let $v \in \mathbf{V}$. v is* extractable *from $d \in \mathbf{D}$ according to $T \in \mathbf{T}$ if: (i) v has type T, and (ii) there exists $d' \preceq d$ such that $sd(v) = d'$.*

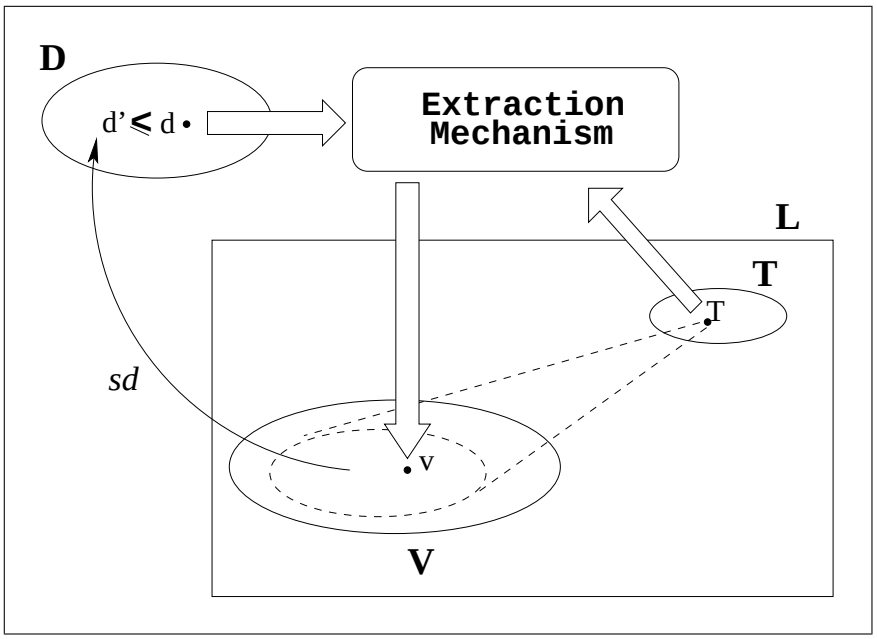

Figure 1. An extraction mechanism for **L**

Finally, an *extraction mechanism* for **L** takes both a document d and a type T, and returns a value v extractable from d according to T, if one exists.

An extraction mechanism is invoked by a programmer, via some interface to the language. If no extractable value can be returned, the programmer is notified of the failure. In case of successful extraction, the returned value may still not fully satisfy the programmer's requirements.

The reason is that, when passing a type to the mechanism, the programmer may not be aware of the exact structure of the target document. Such a type will probably be defined after an eye-inspection of the document or, if available, a description of its structure. As a result, the type may be an under-specification or an over-specification of the data that are potentially relevant to the programmer.

These sort of misjudgements may be very frequent, especially in correspondence with large documents in which data have been inserted at different times and possibly by different users with a different cognition of data representation.

In general, we require that some quantification of relevance be always returned along with the value extracted by the mechanism, and Section 6 will present one quantification scheme in more detail. By interpreting the quantification, the programmer may conclude the inadequacy of the proposed type, refine it, and then re-invoke the mechanism in a prototyping fashion.

1.3. Outline

In the rest of the paper, we concentrate on the formal definition of an extraction mechanism. This is done in Section 4, after the definition of a simplified syntax for XML data and a typed language core in Section 2 and Section 3, respectively.

Section 5 completes the definition of the extraction mechanism by presenting an algorithm that implements it. The algorithm is then used to illustrate a sample quantification scheme in Section 6. Section 7 presents a prototype implementation of the mechanism, while Section 8 and Section 9 examine related work, draw conclusions, and outline further work.

2. A DOCUMENT SYNTAX

In this Section, we define a syntax for XML documents that isolates the data-oriented features of the format (e.g. naming and nesting) from its document-oriented features (e.g. ordering, attributes, processing instructions, etc.). In practice, element attributes may be replaced by subelements.

Definition 2.1 *Let* $\mathbf{D}$ *be the language of* documents *defined by the following grammar:*

$d,\ d_1,\ d_2 ::= <> \mid\ s\ \mid < l \rightarrow d > \mid\ d_1 \wedge d_2$

where $l \in Lbl$*,* $s \in Str$*, and the sets* Lbl *of labels and* Str *of strings are pre-defined languages over the same alphabet of characters.*

A document d is atomic or complex. An atomic document is either the *empty document* $<>$ or else a string. A complex document is either the singleton document $< l \rightarrow d >$ or the concatenation $d_1 \wedge d_2$ of two complex or empty documents. Finally, we shall consider equal two documents that differ only in the ordering of components (e.g. $d_1 \wedge d_2 = d_2 \wedge d_1$).

Essentially, we interpret a document as an edge-labelled tree, in slight contrast with the standard interpretation of XML documents as node-labelled trees. This choice allows us to simplify the formal treatment but has no impact on the applicability of our results.

In the rest of the paper, we shall abbreviate $d =< l_1 \rightarrow d_1 > \wedge \ldots \wedge < l_p \rightarrow d_p >$ with $< l_1 \rightarrow d_1, \ldots, l_p \rightarrow d_p >$ and refer to $l \rightarrow d$ as to an *l-field* with name l and value d. We will also use the function FV_l which takes a document and returns the set of its l-field values.

Definition 2.2 *Let* $l \in Lbl$*. Let* $FV_l : \mathbf{D} \rightarrow \wp(\mathbf{D})$ *be the function:*

$$FV_l(d) = \begin{cases} \{d'\} & d = [l \rightarrow d'] \\ FV_l(d_1) \bigcup FV_l(d_2) & d = d_1 \wedge d_2 \\ \emptyset & otherwise \end{cases}$$

As an example, the following document d

$< member \rightarrow < name \rightarrow Richard,\ age \rightarrow <> > >$

is a rewriting of the XML syntax

```
<member>
   <name>Richard</name>
   <age/>
</member>
```

while $FV_{\texttt{member}}(d) = \{< name \rightarrow Richard, age \rightarrow <> >\}$.

3. A LANGUAGE CORE

In this Section, we present the language **L** for our sample extraction mechanism of Section 4. For our purposes, it suffices a language core defined around a value notation, a language of structural types, and a relationship of typing between the two. Extensions to full-fledged languages with value operators or object-oriented types do not present particular problems.

Along the way, we follow a principle of generality that allows extraction mechanisms for other typed languages to be derived from **L**'s.

We have chosen for **L** a selection of the type constructs commonly found in existing programming languages. Available types are constructed from a range of atomic types $B_1, B_2, \ldots, B_N$. They include record, set, and union types, possibly recursively defined. Record constructors are the singleton record type $[l : T]$ and the concatenation $T_1 \wedge T_2$ of two disjoint record types, where two record types are disjoint when they have no field name in common. Set types are denoted by $set(T)$, where T is the member type of the set. Note that we could have chosen bag types that, differently from set types, can describe repeated sub-documents of a given document. The extension does not present particular problems but it slightly complicates the formal treatment and has been avoided here. Recursive types are denoted by $\mu X.T$, where T is the body of the type and X a type variable, and union types by $T_1 \vee T_2$, where T_1 and T_2 are the branch types of the union. Contrary to most languages, union types are untagged, as we do not need tags to describe alternatives in the structure of the data – of course, this does not exclude the use of tagged unions.

Definition 3.1 *Let* **T** *be the language of* types *generated by the following grammar:*

$$T, T_1, T_2 ::= B_k \mid X \mid [l : T] \mid T_1 \wedge T_2 \mid T_1 \vee T_2 \mid set(T) \mid \mu X.T$$

where $l \in Lbl, X \in Var, k \in [1, n]$.

As for documents, two types are equal if they differ only in the ordering of the components (e.g. $T_1 \wedge T_2 = T_2 \wedge T_1$ and $T_1 \vee T_2 = T_2 \vee T_1$). Similarly, we shall implicitly extend to types the abbreviations, conventions, and auxiliary functions introduced for documents.

The values of **L** include the elements of the atomic types, singleton records $[l = v]$, disjoint concatenations $v_1 \wedge v_2$ of two record values, sets $\{v_1, \ldots v_n\}$, and the empty set $\{\,\}$. To improve readability, we do not syntactically distinguish the operations of concatenation of documents, types, and values. The context shall clarify the domains of definition. In the following examples, we shall assume that **L** include integer numbers n of type *int* and strings "s" of type *string* among its atomic values and types. Finally, value equality follows the same rules as document and type equality.

Definition 3.2 *Let* **V** *be the language of* values *generated by the following grammar:*

$$v, v_1, \ldots, v_n ::= b_k \mid [l = v] \mid v_1 \wedge v_2 \mid \{\,\} \mid \{v_1, \ldots, v_n\}$$

where $l \in Lbl$ *and* $b_k \in B_k$.

The relation of typing between values and types is standard and can be defined as follows:

Definition 3.3 *Let $d \in \mathbf{D}, T \in \mathbf{T}$. d* has type *T if $d : T$, where $: \subseteq \mathbf{D} \times \mathbf{T}$ is the* typing relation *inductively defined by the following rules:*

$$b_k : B_k \quad (ATM) \qquad \{\,\} : set(T) \quad (ESET)$$

$$\frac{v_1 : T, \ldots, v_n : T}{\{v_1, \ldots, v_n\} : set(T)} \quad (SET) \qquad \frac{v : T}{[l = v] : [l : T]} \quad (SREC)$$

$$\frac{v_1 : T_1 \quad v_2 : T_2}{v_1 \wedge v_2 : T_1 \wedge T_2} \quad (REC) \qquad \frac{v : T\left[\mu X.\ T/X\right]}{v : \mu X.T} \quad (RCS)$$

$$\frac{v : T_1}{v : T_1 \vee T_2} \quad (ULFT) \qquad \frac{v : T_2}{v : T_1 \vee T_2} \quad (URGT)$$

where $T\left[\mu X.\ T/X\right]$ denotes the standard operation of (capture-avoiding) variable substitution

4. AN EXTRACTION MECHANISM

In this Section, we provide a self-describing interpretation of language values as well as an inclusion relation between documents.

Definition 4.1 *Let $sd : \mathbf{V} \to \mathbf{D}$ be the function defined as:*

$$sd(b_k) = \texttt{textify}(b_k)$$

$$sd([l = v]) = \begin{cases} <> & v = \{\,\} \\ \bigwedge_{i=1}^{n} < l \to sd(v_i) > & v = \{v_1, \ldots\ v_n\} \\ < l \to sd(v) > & otherwise \end{cases}$$

$$sd(v_1 \wedge v_2) = sd(v_1) \wedge sd(v_2)$$

where `textify` *is any function that returns string representations of atomic values.*

The interpretation is straightforward, except perhaps for the case of set values, which are interpreted only within record values. The reason is that set values are not directly supported in **D** and must be interpreted in correspondence with repeated field names within complex documents.

For example, the document

$<member \to< name \to Steve >,$
$\ member \to< name \to Fabio >>$

interprets the value

$[member = \{[name = \text{“}Steve''], [name = \text{“}Fabio'']\}]$.

In particular, values such as $\{1, 2\}$, $v \vee \{1, 2\}$, or $[a : \{\{1, 2\}, 3\}]$ cannot be interpreted in $\mathbf{D}$ for no field label is available for their interpretation.

Inclusion of documents is susceptible of different interpretations. Here, we have followed the simple intuition according to which d' is included d if d' is syntactically contained in d, with the exception that the empty document is included in any document.

Definition 4.2 *Let $d, d' \in \mathbf{D}$. d' is* contained *in d if $d' \preceq d$, where $\preceq : \subseteq \mathbf{D} \times \mathbf{D}$ is the relation inductively defined by the following rules:*

$$<> \preceq d \qquad (EMP) \qquad\qquad s \preceq s \qquad (STR)$$

$$\frac{d_1 \preceq d_2}{< l \rightarrow d_1 > \preceq < l \rightarrow d_2 >} \quad (SDOC) \qquad\qquad \frac{d_1 \preceq d_3 \quad d_2 \preceq d_4}{d_1 \wedge d_2 \preceq d_3 \wedge d_4} \quad (DOC)$$

5. EXTRACTION ALGORITHM

In this Section, we show the algorithm `Ext` that implements the extraction mechanism defined in Section 4.

Definition 5.1 *Let $\mathbf{V}_\perp = \mathbf{V} \bigcup \{\perp\}$. Let $\texttt{Ext} : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{V}_\perp$ be the algorithm defined as*

```
Ext(d, T) =

1  if unf(T)={T}
2    case of T
3      B_k:
4         { if ∃b_k s.t.  textify(b_k)= d
                  return b_k
5         return ⊥ }

6      T_1 ∧ T_2:  return (Ext(d,T_1)∧_⊥Ext(d,T_2))

7      [l : T'], T' = set(T''):
8         {if d ∈ Str return ⊥
9         A := ∅
10        for each d' ∈ FV_l(d) {
11          v = Ext(d', T'')
12          if v ≠⊥ A = A⋃{v} }
13        return [l=collect(A)] }

14     [l : T'], T' ≠ set(T''):
```

```
15        { for each d' ∈ FV_l(d) {
16          v = Ext(d', T')
17          if v ≠⊥ return [l=v] }
18        return ⊥ }

19     otherwise:  return ⊥

20  else
21   { for each T' ∈ unf(T) {
22     v = Ext(d, T')
23     if v ≠⊥ return v }
24   return ⊥ }
```

where $\wedge_\perp : \mathbf{V}_\perp \times \mathbf{V}_\perp \to \mathbf{V}_\perp$ *is the function defined as:*

$$v_1 \wedge_\perp v_2 = \begin{cases} \perp & v_1 = \perp \text{ or } v_2 = \perp \\ v_1 \wedge v_2 & \text{otherwise} \end{cases}$$

Given $d \in \mathbf{D}$ and $T \in \mathbf{T}$, `Ext` performs a recursive analysis of both type and document and it either fails (i.e. returns $\perp$) or else derives a value $v \in \mathbf{V}$ extractable from d according to T. To achieve this, `Ext` solves two main problems.

The first is that set values must be extracted along with record values, for the same reason underlying Definition 4.1. This explains why `Ext` processes set types only when processing record types and fails with types such as $set(T) \vee T$ or $set(set(T))$.

The situation is complicated further by the possibility that set types do occur within record types but are 'protected' by union or recursive types. Given the type $[a : set(T) \vee T]$, for example, we cannot recursively delegate to the union case the extraction of a value of type $set(T)$, for we would lose the label a necessary to extract the value set from the document.

`Ext` solves the problem by extracting values only according to record types that are 'flattened', i.e. contain no union or recursive type fields. This is achieved with the preliminary check on line 1, which invokes an implementation of the function unf. unf is a generalisation of the standard operation of one-step unfolding of recursive types. In particular, it one-steps unfolds a union type into the set of its branches, and a record type into the set of records obtained by one-step unfolding all its non-record field values.

Definition 5.2 *Let* $unf : \mathbf{T} \to \mathbf{T}$ *be the function defined as:*

$$unf(T) = \begin{cases} \{T' \left[\mu X.\ T/X\right]\} & T = \mu X.T' \\ \{T_1, T_2\} & T = T_1 \vee T_2 \\ \{[l : T'] \mid T' \in unf_R(T)\} & T = [l : T] \\ \{T_1' \wedge T_2' \mid \\ \quad T_1' \in unf(T_1), T_2' \in unf(T_2)\} & T = T_1 \wedge T_2 \\ \{T\} & \text{otherwise} \end{cases}$$

where $unf_R : \mathbf{T} \to \mathbf{T}$ *is the function*

$$unf_R(T) = \begin{cases} \{T\} & T = [l : T] \\ unf(T) & \text{otherwise} \end{cases}$$

For example,

$$unf([a : int \vee set(string), b : [c : int \vee set(int)]]) = \{\ [a : int, b : [c : int \vee set(int)]],\ [a : set(string), b : [c : int \vee set(int)]]\}.$$

A type T is then *unfolded* if $unf(T) = \{T\}$, otherwise is *folded*.

The second problem occurs when multiple values of type T can be extracted from d. Due to the presence of set and union types, this possibility is in fact the norm. For example, consider the document $d =< a \to 1, a \to 2, b \to 3, c \to four >$ and the type $T = T_1 \vee T_2$, where $T_1 = [a : set(int), b : int]$ and $T_2 = [a : set(int), c : string]$.

¿From Definition 1.1 and Definition 4.1, it is easy to see that the values $[a = \{1, 2\}, b = 3]$ and $[a = \{1, 2\}, c = "four"]$ are extractable from d according to T_1 and T_2, respectively, and thus according to T. The same is true of the values $[a = \{\ \}, b = 3]$, $[a = \{1\}, b = 3]$, $[a = \{2\}, b = 3]$, etc.

`Ext` returns one extractable value on the basis of both a *best-attempt* and a *first-attempt* policy. Specifically, it returns: (i) the largest value extractable from d according to a set type, and (ii) the first value extractable from d according to one of the branches of union types, when these are ordered from left to right (left-to-right is also the branch ordering followed by the sub-routine `unf`). In the previous example, `Ext` derives the value $[a = \{1, 2\}, b = 3]$.

The best-attempt policy is justified by an immediate *principle of maximisation* of the data contained in a correctly extracted value. The first-attempt policy is instead more arbitrary but it simplifies the definition of the algorithm and is thus adequate for the purpose of a proof of concept.

In lines 21-23, `Ext` implements its first-attempt policy and either fails or returns the first extractable value returned by a recursive execution of `Ext` with d and an unfolding of T.

In lines 2-19, `Ext` processes basic, singleton record, and concatenated record types. For a basic type B_k, `Ext` is successful only if d 'textifies' a value v of type B_k (lines 3-5).

For singleton record types $[l : T']$, `Ext` tries to derive a singleton record value $[l : v]$, where the shape of v depends on whether T' is a set type.

If T' is a set type $set(T'')$ (lines 7-13), `Ext` implements its best-attempt policy and tries to extract a value of type T'' from each l-field value of d. The extracted values are then memorised and eventually grouped into a set value v by the sub-routine `collect`. Notice that, in this case, `Ext` fails only when the document is a string (line 8). Otherwise, it returns at worst the singleton record $[l = \{\ \}]$.

If T' is not a set type (lines 14-18), `Ext` returns the first value of type T' extractable from an l-field value of d. If such value does not exist, `Ext` fails.

If T is the concatenation of two record types T_1 and T_2, `Ext` concatenates the values extracted from d according to T_1 and T_2 respectively. The operation of concatenation $\wedge_\perp$ refines standard concatenation by returning $\perp$ any time one of the operands is $\perp$. This ensures that failing the extraction according to either singleton record type fails the entire process.

`Ext` is clearly terminating, for it recursively operates on subdocuments of the input document and because `unf`, `textify`, and `collect` are trivially terminating. Note that termination holds under the standard assumption that contractive recursive types, such as $\mu X.X$, are not part of the type language (cf. [10]). Next, we shall also prove that `Ext` is sound, i.e. returns a value extractable from d according to T, and complete, i.e. it fails only when no value of type T can be extracted from d.

This completeness result is already sufficient to give users confidence in applications of `Ext`. As shown earlier, however, the first-attempt policy of the algorithm and the presence of union types prevents the user of assuming any relationship between `Ext`'s output and any other value of type T extractable from d. In some cases, this may not be satisfactory. When executed against the document $< a \rightarrow 1, b \rightarrow 2, c \rightarrow foo >$ and the type $[a : int] \vee [a : int, c : string]$, for example, `Ext` returns the value $[a = 1]$ instead of the 'larger' $[a = 1, c = "foo"]$.

5.1. Correctness

Lemma 5.3 *Let $v \in \mathbf{V}$,$T \in \mathbf{T}$. $v : T$ if and only if there exists $T' \in unf(T)$ such that $v : T'$.*

Proof. By structural induction on $\mathbf{T}$. The proof is immediate and we shall here discuss only the cases $T = T_1 \wedge T_2$.

Assume $v : T$. For typing scheme (REC), $v = v_1 \wedge v_2$, with $v_1 : T_1$, and $v_2 : T_2$. For the inductive hypothesis, $v_1 : T_1'$ and $v_2 : T_2'$, for some $T_1' \in unf(T_1)$ and $T_2' \in unf(T_2)$. For typing scheme (REC) and Definition 5.2, $v = v_1 \wedge v_2 : T_1' \wedge T_2' \in unf(T_1 \wedge T_2) = unf(T)$.

Vice versa, assume $v : T'$, with $T' \in unf(T)$. For Definition 5.2, $T' = T_1' \wedge T_2'$, with $T_1' \in unf(T_1)$ and $T_2' \in unf(T_2)$. For type scheme (REC), $v = v_1 \wedge v_2, v_1 : T_1', v_2 : T_2'$. For the inductive hypothesis and typing scheme (REC), $v = v_1 \wedge v_2 : T_1 \wedge T_2 = T$.

◇

Proposition 5.4 *Let $d \in \mathbf{D}$, $T \in \mathbf{T}$. If* `Ext`$(d, T) \neq \perp$ *then* `Ext`(d, T) *is extractable from d according to T.*

Proof. By induction on the height h of the execution tree of `Ext`(d, T).

For the hypothesis, the case $h = 1$ corresponds to one of the cases $T = B_k$ and $T' = [l = T']$, with $T = set(T'')$. In the first case, `Ext`$(d, T) = b_k$ and $sd(b_k) = d$. The thesis follows then from typing scheme (ATM), and the reflexivity of document inclusion, which can be easily proven by structural induction on $\mathbf{D}$. In the second case, $v = [l = \{\,\}]$. For Definition 4.1 and inclusion scheme (EMP), $sd(v) = sd([l = \{\,\}]) = <> \preceq d$. For typing schemes (SREC) and (ESET), $v : T$ and the thesis is proven.

Assume now an execution tree of height $h > 1$ and that the thesis is proven for all execution trees of height $h - 1$. Let us distinguish the cases in which T is folded or unfolded.

If T is folded, the hypothesis ensures that `Ext`(d, T)=`Ext`$(d, T') = v \neq \perp$ for some $T' \in unf(T)$. For the inductive hypothesis, $sd(v) \preceq d$ and $v : T'$. For Lemma 5.3, $v : T$ and the thesis is proven.

If T is unfolded, there are three cases to examine:

(*i*) $T = T_1 \wedge T_2$. For the hypothesis and the definition of $\wedge_\perp$, $\texttt{Ext}(d, T) = v = v_1 \wedge v_2$, where $v_1 = \texttt{Ext}(d, T_1) \neq \perp$ and $v_2 = \texttt{Ext}(d, T_2) \neq \perp$. For the inductive hypothesis, $sd(v_1) \preceq d$,$v_1 : T_1$,$sd(v_2) \preceq d$, and $v_2 : T_2$. For Definition 4.1 and inclusion scheme (DOC) , $sd(v) = sd(v_1 \wedge v_2) = sd(v_1) \wedge sd(v_2) \preceq d \wedge d = d$. For typing scheme (REC), $v : T$ and the thesis is proven.

(*ii*) $[l : T']$, with $T' = set(T'')$. For $h > 1$, $FV_l(d) \neq \emptyset$. For the hypothesis, $\texttt{Ext}(d, T) = v = [l = \{v_1, \ldots, v_n\}]$, where $v_i = \texttt{Ext}(d_i, T'') \neq \perp$, $d_i \in FV_l(d)$, for each $i \in [1, n]$ and some $n \in N$. For the inductive hypothesis, $sd(v_i) \preceq d$ and $v_i : T''$. For Definition 4.1 and scheme (DOC) , $sd(v) = sd([l = \{v_1, \ldots, v_n\}]) = \bigwedge_{i=1}^{n} < l \rightarrow sd(v_i) > \preceq \bigwedge_{i=1}^{n} < l \rightarrow d_i > \preceq d$. For typing schemes (SREC) and (SET), $v : T$ and the thesis is proven.

(*iii*) $[l : T']$, with $T' \neq set(T'')$. For the hypothesis, $\texttt{Ext}(d, T) = v = [l = v']$, where $v' = \texttt{Ext}(d', T') \neq \perp, d' \in FV_l(d)$. In particular, $d = [l : d'] \wedge d''$, for some $d'' \in (D)$. For the inductive hypothesis, $sd(v') \preceq d'$ and $v' : T'$. For Definition 4.1 and inclusion scheme (DOC), $sd(v) = sd([l = v']) =< l \rightarrow sd(v') > \preceq d$. For scheme (SREC), $v : T$ and the thesis is proven.

$\diamond$

Proposition 5.5 *Let $d \in \mathbf{D}$, $T \in \mathbf{T}$, $v \in \mathbf{V}$. If v is extractable from d according to T then $\texttt{Ext}(d, T) \neq \perp$.*

Proof. By induction on the height h of the proof tree of $v : T$.

The case $h = 1$ corresponds to one of the case $v = b_k$ and $v = \{\,\}$. The second case is excluded by the hypothesis and Definition 4.1. In the first case, $T = B_k$, $\texttt{Ext}(d, T) = b_k \neq \perp$ and the thesis is proven.

Assume now an execution tree of height $h > 1$ and that the thesis is proven for all execution trees of height $h - 1$. Let us distinguish the cases in which T is folded or unfolded.

If T is folded, the hypothesis $v : T$ and Lemma 5.3 ensure that $v : T'$, for some $T' \in unf(T)$, and thus that v is extractable from d according to T'. For the inductive hypothesis, $\texttt{Ext}(d, T') \neq \perp$ and thus $\texttt{Ext}(d, T) \neq \perp$.

If T is unfolded, there are three cases to examine.

(*i*) $T = T_1 \wedge T_2$. For typing scheme (REC) there exist $v_1, v_2 \in \mathbf{V}$ such that $v_1 : T_1$, $v_2 : T_2$, and $v = v_1 \wedge v_2$. For the definition of $\wedge$, Definition 4.1, inclusion schemes (EMP) and (DOC), and the hypothesis $sd(v) \preceq d$, $sd(v_1) = sd(v_1) \wedge <> \preceq sd(v_1) \wedge sd(v_2) = sd(v_1 \wedge v_2) = sd(v) \preceq d$. Similarly, $sd(v_2) \preceq d$. For the inductive hypothesis, $\texttt{Ext}(d, T_1) \neq \perp$ and $\texttt{Ext}(d, T_2) \neq \perp$. For the definition of $\wedge_\perp$, $\texttt{Ext}(d, T) = \texttt{Ext}(d, T_1) \wedge_\perp \texttt{Ext}(d, T_2) \neq \perp$, and the thesis is proven.

(*ii*) $[l : T']$, with $T' = set(T'')$. The thesis follows immediately from $\texttt{Ext}(d, T) = [l = v']$, for some $v' \in (V)$.

(*iii*) $[l : T']$, with $T' \neq set(T'')$. For the hypothesis $v : T$ and scheme (SREC), $v = [l = v']$, and $v' : T'$. For the hypothesis $sd(v) \preceq d$ and Definition 4.1, $sd(v) = sd([l =$

$v'] =< l \rightarrow sd(v') \preceq d$. From inclusion scheme (DOC), $d = [l : d'] \wedge d''$, for some $d', d'' \in (D)$, where $sd(v') \preceq d'$. For the inductive hypothesis, $\texttt{Ext}(d', T') = v'' \neq \perp$, $\texttt{Ext}(d', T') = [l : v''] \neq \perp$, and the thesis is proven.

$\diamond$

6. RELEVANCE

In this Section, we present a simple relevance quantification scheme for the extraction mechanism defined in Section 4. The scheme can be easily embedded in `Ext`, but we give it here a separate specification to improve readability.

Let us start with a motivating example. Consider the following document d:

$$
\begin{array}{l}
< staff \\
\quad < member \rightarrow \\
\quad\quad < name \rightarrow David > \\
\quad\quad < project \rightarrow \\
\quad\quad\quad < name \rightarrow SNAQue, \ldots > \ldots > \\
\quad\quad < project \rightarrow \\
\quad\quad\quad < name \rightarrow GLOSS, \ldots > \ldots > \ldots > \\
\quad < member \rightarrow \\
\quad\quad < name \rightarrow Paolo > \\
\quad\quad < project \rightarrow \\
\quad\quad\quad < name \rightarrow Tequyla, \ldots > \ldots > \\
\quad\quad < project \rightarrow \\
\quad\quad\quad < name \rightarrow TQL, \ldots > \ldots > \ldots > \ldots >
\end{array}
$$

and assume that the programmer requires to compute over named projects of staff members.

¿From an initial analysis of d, the programmer proposes the type T:

$$staff : [member : set([project : [name : string]])]]$$

and the mechanism returns the value v:

$$
\begin{array}{l}
[staff = [member = \{[project = [name = "SNAQue"]], \\
\qquad\qquad\qquad\qquad [project = [name = "TeQuyLa"]]\}]].
\end{array}
$$

The programmer did not notice that staff members have more than one project, i.e. d contains more relevant data than T makes possible to extract.

To inform the user, we quantify the *precision* with which T describes the data in d that are relevant to the programmer. To return a readable measure, we distribute it along the singleton record types occurring in T, i.e where loss of relevant data may actually occur. The result is a set of annotations for T that may help the user to refine the type and improve extraction.

In particular, the precision of a singleton record type $[l : T']$ is measured with respect to all the documents that are processed with $[l : T']$ on a successful execution path of

$\texttt{Ext}(d, T)$ (by successful execution path of $\texttt{Ext}(d, T)$, we intend a path of the execution tree of $\texttt{Ext}(d, T)$ along which Ext never fails).

Let thus $D_{[l:T']}$ be the set of all such documents, and let $P_{[l:T']}$ be the set defined as:

$$P_{[l:T']} = \{\, (d', [l = v']) \mid d' \in D_{T'},\ [l = v'] = \texttt{Ext}(d', [l : T'])\,\}.$$

The precision $prec_{[l:T']}$ of $[l : T']$ is then calculated as:

$$prec_{[l:T']} = \frac{\sum_{p \in P_{[l:T']}} vprec(p)}{\mid P_{[l:T']} \mid}$$

where the *value precision* $vprec$ of a pair in $P_{[l:T']}$ is defined as:

$$vprec(d', [l = v']) = \begin{cases} \frac{|v'|}{\sigma} & \sigma > 0 \\ 1 & \sigma = 0 \end{cases}$$

and, in turn, $\sigma = \mid FV_l(d') \mid$ and

$$\mid v' \mid = \begin{cases} 0 & v' = \{\,\} \\ n & v' = \{v_1, \ldots, v_n\} \\ 1 & otherwise \end{cases}$$

Informally, $prec_{[l:T']}$ is the average of the precisions calculated at each pair $(d', [l = v']) \in P_{[l:T']}$. Each of these is in turn the ratio between the number of l-field values from which Ext extracted a value and the number of those it did not.

In particular, low precision for a singleton $T_1 = [l : set(T')]$ in T suggests that Ext did not extract values of type T' from many l-field values in d. A renewed analysis of the data may then reveal that the singleton $T_2 = [l : set(T' \vee T'')]$ allows to extract more relevant data from d and should thus replace T_1. Similarly, $[l : set(T')]$ may do better than a low-precision singleton $[l : T']$ if $T' \neq set(T'')$.

For example, the precision of type T in the previous example may be returned as the following annotation:

$$^1[staff\,^1[member : set\{^{\frac{1}{2}}[project :^1 [name : string]]\}]]$$

The user may then improve extraction by refining T into the type:

$$[staff : [member : set\{[project : set([name : string])]\}]]$$

which has precision 1 on all its singleton record types.

The problem of relevance quantification is certainly complex and identifies an interesting research topic per se. The scheme presented is fairly simple, and we have introduced it as a proof of concept. Although we have not yet gathered experimental results, we believe that the scheme can be useful with large datasets, where the exact structure of relevant data is not known when the mechanisms is first invoked.

7. SNAQue

Based on the algorithm `Ext`, we have built a distributed system prototype. The system, maintained at the Computer Science Department of the University of Strathclyde, is called `SNAQue` - *the Strathclyde Novel Architecture for Querying extensible mark-up language* (cf. [11]). Although some parts are still under development, the system is currently being tested in a number of biodiversity projects by the Palaeobiology Research Group at the University of Glasgow.

SNAQue is a CORBA application that implements an extraction mechanism for a subset of the *CORBA Interface Definition Language* (cf. [12]), and therefore for any CORBA-compliant language (e.g. C, C++, Smalltalk, Java, Ada95, etc.).

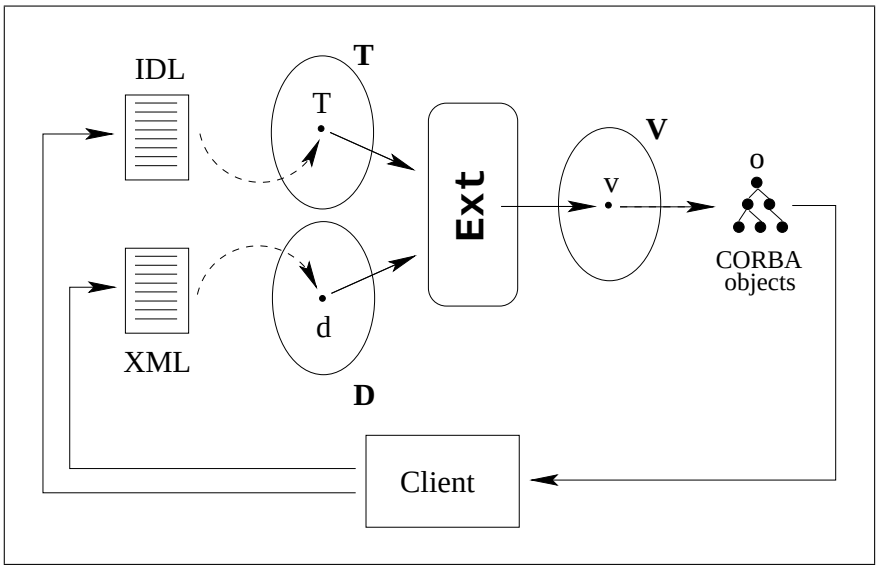

Figure 2. The SNAQue architecture

The correctness of the system relies directly on the correctness of the extraction mechanism defined in Section 4. In particular, SNAQue receives an XML document and an IDL type description from a remote client, and maps them onto a document $d \in \mathbf{D}$ and a type $T \in \mathbf{T}$, respectively. It then invokes `Ext` and transforms the output $v \in \mathbf{V}$ of type T in a number of interrelated CORBA objects. By virtue of the mapping from IDL to $\mathbf{T}$, the objects expose interfaces corresponding to the initial IDL description.

There is always one entry point to the generated CORBA objects: the object o that corresponds to the root of the XML document. A reference to o is returned to the client for local binding in programs written in any CORBA-compliant language of choice. Optionally, SNAQue may register o with a public alias provided by the client. Then any client informed of the alias can come along and gain remote access to o (see Figure 2).

The details of the mappings from XML to $\mathbf{D}$, IDL to $\mathbf{T}$, and $\mathbf{V}$ to the corresponding CORBA objects are out of the scope of this paper. Roughly, the mapping from XML

concentrates on the data-oriented features of the format, while the mapping from IDL converts interfaces, sequences, and tagged union types, into records, sets, and untagged union types in **T**.

As an example, consider the document d and its fragment d' introduced in Section 1. Using SNAQue, it is straightforward to compute over d' with a code analogous to that shown in Section 1. One has only to provide SNAQue with the following IDL type description:

```
interface Staff {
   typedef sequence <Member> MemberSeq;
   attribute MemberSeq members; }

interface Member {
   attribute String name;
   attribute long code; }
```

With respect to these inputs, SNAQue will create four CORBA objects: one conforming to the `Staff` interface and three conforming to the `Member` interface. It will then return a reference to the first object to which clients can bind in their programs.

SNAQue choses Java to implement the extracted CORBA objects. In particular, the Java classes generated by the system for the objects derived above are exactly the `Staff` and `Member` classes shown in Section 1. Similarly, the Java-like code suggested there could be immediately used to compute over the data.

The choice of CORBA IDL as the type language is an obvious one. It increases the applicability of the extraction mechanism for **L** and makes it distributed. However, the distributed nature raises performance issues, such as those related with every read or write operation performed on the data across the network.

We are currently investigating two ways of tackling such problems. On the one hand, we are considering the use of *value types*, which have been recently introduced into CORBA to allow objects to be passed by value, rather than by reference. This allows clients to *pull* the generated CORBA objects over the network and inject them in the local environment. On the other hand, we could push computations to the server, by allowing clients to specify additional methods in the IDL interfaces. For example, the `Staff` interface could be extended with a new method:

```
interface Staff {
   typedef sequence<Member> MemberSeq;
   attribute MemberSeq members;
   Member getMember(in String name) }
```

SNAQue can not automatically generate the implementation for this method, which has to be provided by the client. Due to the lack of space, we cannot discuss this facility in more detail.

8. RELATED WORK

The differences between extraction mechanisms and existing approaches has been largely discussed in Section 1. It is worth noticing here that extraction mechanisms operate on arbitrary XML documents and can thus be easily coupled with untyped query languages. In an integrated environment, the convenience of the first would complement the flexibility of the second for data with a varying degree of structural regularity.

Other high-level bindings between XML and existing programming languages have been recently presented (cf. [13–15]. They all map some form of data description (usually a DTD or an XML Schema) onto language types that capture directly the semantics intended for the data. For this reason, they operate on fairly regular XML documents and do not provide facilities for extracting regular subsets from arbitrary documents. In addition, they have been developed for specific languages (e.g. Haskell, Java) and do not generalise.

The idea of exploiting regularity in XML, and more generally, semistructured data, has also motivated a number of approaches.

Early proposals were for extending standard database technology to accommodate some degree of irregularity in the data, typically via the provision of union types. Although similar in motivation, such approaches differ from ours in their attempt to provide a total description of the data. As mentioned in Section 1, significantly irregular data lead to an uncontrolled use of union types, thereby progressively decreasing system performance and complicating program specification.

Later proposals assume a dedicated query language as a starting point, but differ in their data-first or type-first strategy.

Approaches of the first kind infer type information from existing datasets. In this case, type inference can be performed by the system for the entire database, automatically or semi-automatically (cf. [16–20]). The resulting types are mainly for users to understand the data and, to some extent, for query optimisers to improve execution (cf. [21]). Partial inference can be also performed by users, and the results then fed to the system as hints to reduce the scope of a search (cf. [22]). Overall, inference-based approaches exploit typing for resource optimisation, while computations remain essentially untyped.

Approaches of the second kind exploit static knowledge to guarantee computational safety (cf. [23–26]). To achieve this against a tree-based model, they resort to low-level types for XML documents. Due to the support of regular expressions, such *tree types* are more flexible than high-level types in capturing irregularities in the data (cf. [27–30]).

To the best of our knowledge, Ozone (cf. [31]) is the only attempt to seamlessly integrate structured and semistructured data in the same typed environment. The system extends the ODMG model to include semistructured data, and allows structured objects to be queried with semistructured primitives. Interestingly, it also supports a function for coercing semistructured data to structured objects according to a type and, as such, implements a simple extraction mechanism for ODMG. However, our mechanism is proved correct and returns values of a larger set of types.

9. CONCLUSIONS AND FUTURE ISSUES

We have presented a novel approach to programming over XML data based on language bindings. The bindings are defined as mechanisms that identify and derive language values from subsets of arbitrary XML documents. When programming over such subsets, the approach delivers the computational advantages associated with the host language. Furthermore, the derived values preserve the semantics intended for the data, and thus facilitate program specification.

These mechanisms can be formally defined and correctly implemented, and we have done it for a sample but representative core language. In particular, we have proven the generality of the sample mechanism by deriving extraction mechanisms for all CORBA-compliant languages directly from it.

Future research directions concern both theoretical and practical aspects of the investigation. Beyond XML, we have already extended our results to more general forms of semistructured data. In particular, we are able to extract language values from graph-structured data, i.e. in the presence of cycles and sharing. The interested reader is referred to [32] for the full treatment.

Another interesting direction relates to the definition of inclusion between documents. The one we proposed follows first intuitions, but alternative definitions could be considered. As a first example example, inclusion checks may start from arbitrary elements of the target document, not necessarily the root element. This would save the user the often tedious task of describing the structure that leads from the root of the document to the data of interest. It would also give a hint of the flexibility achieved by navigational query paradigms without reducing the advantages of the approach.

The extraction algorithm has been proved sound but the belief that it is also tractable in pragmatic terms has not been supported by a formal analysis of its complexity. Although tests on large data samples have shown acceptable performance even on desktop machines, the impact of a considerable use of union types remains to be measured. Furthermore, we are currently working with back-tracking techniques towards algorithms with stronger properties of completeness.

Relevance quantification could be certainly improved over the sample scheme proposed in Section 6. In particular, an extraction mechanism could customise a general scheme to the programmer's specifications.

Finally, SNAQue is under continuous development, and a web interface to the system is being published at the time of writing. The research agenda is currently focusing on whether values can be virtually injected in the value space of the target language rather than materialised. Single or multiple indexes to regular subsets suggest the possibility to dynamically synchronise the interface between the language and the database under updates. At the same time, they raise the opportunity for incremental extractions.

In addition, the client/server scenario raises a number of questions related to the efficiency of the system and to the possibility of integrating data from distributed XML servers.

Investigation is needed to identify the cases in which it is more convenient to pull extracted values at the client side or else push client computations to the server. Com-

pleteness quantification could here be used by both client and server to make intelligent decisions about data or code migration.

In addition, the possibility of storing and publishing typed interfaces over the data at the server side suggests interesting data protection and data evolution policies. For example, the usage and volume of XML data referenced through the interfaces could be gathered into statistical information that may be used to assess the impact of changes to the data.

REFERENCES

1. T. Bray, J. Paoli, C. Sperberg-McQueen, Extendible Markup Language (XML) 1.0, Tech. rep., World Wide Web Consortium, w3C Recommendation (1998).
2. S. Abiteboul, Querying semi-structured data, Database Theory - ICDT '97, 6th International Conference, Delphi, Greece (1997) 1–18.
3. P. Buneman, Semi-structured data, In Proceedings of the Sixteenth ACM SIGACT - SIGMOD – SIGART Symposium on Principles of Database Systems (1997) 117–121.
4. P. Buneman, M. Fernandez, D. Suciu, UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion, VLDB Journal 9 (1) (2000) 76–110.
5. S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, The Lorel Query Language for Semistuctured Data, Journal of Digital Libraries, 1(1) (1997) 68–88.
6. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, XML-QL: A Query Language for XML, Tech. rep., World Wide Web Consortium, submission to the World Wide Web Consortium (Aug. 1998).
7. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, Lore: a database management system for semistructured data, in: SIGMOD Records, 26(3), 1997, pp. 54–66.
8. Document object model (DOM), http://www.w3.org/DOM.
9. M. T. Ltd., SAX 2.0: The Simple API for XML, http://megginson.com/SAX/ (2000).
10. G. Amadio, L. Cardelli, Subtyping recursive types, ACM Transactions on Programming Languages and Systems(TOPLAS), 15(4) (1993) 575–631.
11. A. Stavrianou, Querying XML through a CORBA gateway, Master's thesis, University of Glasgow (UK) (September 1999).
12. OMG, CORBA OMG IDL text file - The Object Management Group, ftp://ftp.omg.org/pub/docs/formal/99-04-01.txt (1999).
13. M. Wallace, C. Ranciman, Haskell and XML: Generic combinators or type-based translation?, in: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99), Vol. 34-9 of ACM Sigplan Notices, ACM Press, N.Y., 1999, pp. 148–159.
14. T. B. Factor, Xml data binding and breeze xml studio (white paper) .
15. S. M. Inc., Web Services Made Easier: The Java APIs for XML .
16. S. Nestorov, J. D. Ullman, J. L. Wiener, S. S. Chawathe, Representative objects: Concise representations of semistructured, hierarchial data, in: A. Gray, P.-Å. Larson (Eds.), Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K, IEEE Computer Society, 1997, pp. 79–90.

17. S. Nestorov, S. Abiteboul, R. Motwani, Inferring structure in semistructured data, Workshop on Management of Semistructured Data, in conjuction with PODS/SIGMOD, Tucson - Arizona .
18. S. Nestorov, S. Abiteboul, R. Motwani, Extracting schema from semistructured data, ACM SIGMOD (1998) 295–306.
19. R. Goldman, J. Widom, DataGuides: Enabling query formulation and optimization in semistructured databases, in: VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece, Morgan Kaufmann, 1997, pp. 436–445.
20. R. Goldman, J. Widom, Approximate DataGuides, in: Proceedings of the second International Workshop WebDB '99, Pennsylvania, 1999.
21. M. Fernandez, D. Suciu, Optimizing regular path expressions using graph schemas (full version), manuscript available from `http://www.research.att.com/~{mff,suciu}` (February 1997).
22. P. Buneman, S. Davidson, M. Fernandez, D. Suciu, Adding structure to unstructured data, Lecture Notes in Computer Science 1186 (1997) 336–350.
23. H. Hosoya, B. C. Pierce, XDuce: A typed XML processing language (preliminary report), webDB workshop (May 2000).
24. A. Albano, D. Colazzo, G. Ghelli, P. Manghi, C. Sartiani, A type system for querying xml documents, in: Proceedings of ACM SIGIR 2000 Workshop On XML and Information Retrieval, Athens, Greece, 2000.
25. D. Chamberlin, D. Florescu, J. Robie, Quilt: an XML query language for heterogeneous data sources, in: Proceedings of WebDB, Dallas, TX, 2000.
26. S. Cluet, C. Delobel, J. Siméon, K. Smaga, Your mediators need data conversion!, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98), Vol. 27,2 of ACM SIGMOD Record, ACM Press, New York, 1998, pp. 177–188.
27. D. C. Fallside, XML Schema Part 0: Primer, Tech. rep., World Wide Web Consortium, w3C Candidate Recommendation (Oct. 2000).
28. H. S. Thompson, D. Beech, M. Maloney, N. Mendelson, XML Schema Part 1: Structures, Tech. rep., World Wide Web Consortium, w3C Working Draft (Dec. 1999).
29. P. V. Biron, A. Malhotra, XML Schema Part 2: Datatypes, Tech. rep., World Wide Web Consortium, w3C Working Draft (Dec. 1999).
30. Arnaud Sahuguet, Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask, in: WebDB-2000, 2000.
31. S. Abiteboul, R. Goldman, T. Lahiri, J. McHugh, J. Widom, Ozone: Integrating structured and semistructured data, Tech. rep., Stanford University Database Group (1998).
32. P. Manghi, Extracting typed values from semistructured databases, Ph.D. thesis, Dipartimento di Informatica, Università di Pisa - Italy (2001).