Language Bindings to XML

By preserving application semantics, XML bindings improve program specification, verification, and optimization. SNAQue makes it all simple and flexible.

Fabio Simeoni,
David Lievens,
and Richard Connor
Strathclyde University, Glasgow, Scotland

Paolo Manghi
Università di Pisa, Italy

XML's significance extends beyond the purpose it was originally intended to serve (document processing over the Web). As a standard and widely supported format for arbitrary data, XML delivers to any heterogeneous, distributed computer system the common lingo for data exchange — something only closed or relatively closed systems could previously assume and enforce. In addition, XML data contains a description of its intended meaning, either in the form of element names or, optionally, as an embedded or referenced type description (for example, a DTD). *Self-description* allows programs to interpret the data dynamically — with no reliance on shared assumptions or agreements (such as header files or interface definition languages), and thus makes them more resilient to changes in the data.[1] In its untyped form, though, XML is an ideal carrier for *semistructured* data, where the structure is too irregular or instable to be effectively handled in statically typed programming languages or database management systems.[2]

This broad view of XML explains the proliferation of related technologies, especially programming models. To date, computations over XML data can be specified in a variety of paradigms and languages. Most approaches, however, have so far focused on novel and dedicated solutions: from flexible query languages, which resort to regular expressions that match data with irregular or partially known structure (for example, XML-QL, XQL, and LOREL) to Turing-complete or strongly typed functional languages, which exploit structural regularity to ensure correctness of arbitrary computations (for example, XQuery, XSLT, and Xduce).

In contrast, the computational facilities of mainstream, statically typed languages have only been partially reused in this novel context (such as the completeness and simplicity of the procedural and object-oriented programming models, the reliability and efficiency of computations, and equally important, the

 Published by the IEEE Computer Society 1089-7801/03/$19.00

large user-base and tool support). This is the domain of *language bindings* to XML — software mechanisms that transform XML data into values that programmers can access and manipulate from within their language of choice. In this article, we compare two standard binding solutions — namely, the Simple API for XML (SAX) and the Document Object Model (DOM) API — and two novel approaches — Sun's JAXB architecture and our Strathclyde Novel Architecture for Querying XML (SNAQue).

A Motivating Example

We identify two broad scenarios in which we could conveniently bridge XML and typed languages. The first approach accounts for most current XML usage — namely, the exchange of business data across proprietary boundaries. The second concerns semistructured documents with large fragments of more disciplined data, such as those that arise in scientific domains or from the integration of heterogeneous databases.

In general, the goal is to compute over encodings of real-world entities that are maintained or even generated outside the language jurisdiction and that occur as fragments of (possibly semistructured) XML documents. For example, consider the following document *d* about a university department's staff members:

```
<staff code="123517">
  <member>
    <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 code="175417">
  <member>
    <ext>4566</ext>
    <name>Fabio Simeoni</name>
  </member>
</staff>
```

For the sake of illustration, we've added a few irregularities to the data: some staff members have Web page information and others have projects or phone extensions. On a much larger scale, these irregularities discourage the use of conventionally typed technologies; although union types and object–oriented hierarchies can accommodate some of this irregularity, their extensive use would soon reduce the system's static knowledge and complicate program specification and maintenance.

Notice, though, that all members in *d* have a name and a code. The following fragment *d′* of *d* could be an XML representation of a standard language value:

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

For most statically typed object–oriented languages, *d′* could be an XML encoding of the state of an object `staff` of class `Staff`, where

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

and

```
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′* should be at least as simple, safe, and efficient as programming over `staff`. Furthermore, we would like these properties to scale — that is, hold for generalized computations over fragments of XML documents considerably larger than *d′*.

Motivated by these requirements, we advocate the importance of language bindings to XML. The host language's computational facilities do not guaran-

tee per se the convenience of a binding, however, as the resulting values must also support the program's interpretation of the data. One aim of this article is to show that when this is not the case, programmers benefit only partially from the host language's expressiveness, strong typing, and efficiency.

SAX and DOM

From a binding perspective, SAX transforms an XML document into a string served to the programmer as a temporal sequence of tokens. The induced programming model is based on parser–generated events and programmer-implemented callbacks.

The SAX programmer is thus invited to share the interpretation of the data with the underlying parser, albeit at a higher level of abstraction. In particular, the programmer receives only tokens that correspond to distinguished components of the format, such as PCDATA sections or element names. The following code represents a SAX program that iterates over staff members to retrieve their names and codes:

```
class SaxTask extends DefaultHandler {
   private String name;
   private int code;
   private boolean inProject;
   private CharArrayWriter buffer
     = new CharArrayWriter();

   public void characters (char[ ] ch,int
         start,int length)
      {buffer.write(ch,start,length);}

   public void startElement (String
         uri,String name,String qName,
         Attributes atts){
      if (name.equals("member"))
        code=atts.getValue("code");
      if (name.equals("project"))
        inProject = true;
      buffer.reset();}

   public void endElement (String
         uri,String name,String qName){
      if (name.equals("project"))
        inProject = false;
      if (name.equals("name") &&
        !inProject) name =
        buffer.toString().trim();
      ...do something with name and
        code...}}
```

The DOM API, on the other hand, represents an XML document as a node–labeled tree in which nodes correspond to the distinguished components of the format. The DOM programmer is thus invited to interpret the data as the document in which it is contained — that is, in terms of the "has–a" relationships between its structural components. The following is a DOM solution to the previous task:

```
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 such a simple task, both programming solutions are quite convoluted. Facing problems on a much larger scale, SAX and DOM programmers have described their code as tedious, hard to read and maintain, and thus prone to errors.

This is due in part to the document–oriented nature of any complete programming interface to XML. Conceptually related data (for example, staff member names and codes) 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 data available). The main problem, however, is that the computation is expressed in an algebra of strings and trees, whereas domain–specific concepts (such as staff names and codes) are relegated to the role of actual parameters.

In SAX, for example, programmers don't ask for a staff member's name; they patiently collect the

characters of `name` elements after they ensure that they're using the right `name` element. In DOM, the programmer would first reach a `member` node, scan its children nodes to get a `name` node, and then eventually get its string value. SAX does not even reflect the structure of the data, so to get things right, the programmer must coordinate possibly distant methods.

Inadequate data abstractions also compromise static checking of computations. Correctness can be guaranteed for generic operations on strings and trees, but not on, say, staff members. The DOM invocation

```
staff.getElementsByTagName("mebmer");
```

where `mebmer` is a typo for `member`, would silently compile and return a null value at runtime. Safety thus becomes the programmer's responsibility, not the system's.

Programmatic checks worsen code readability and maintainability and are not always sufficient to guarantee correct behavior. Lacking a document description, the code could interpret the `mebmer` typo as the absence of required data and thus trigger unintended behavior. Even assuming some kind of document description, the typo could accidentally identify some other data or, in the best case, simply be signaled at runtime.

For similar reasons, the system can optimize resources only within the limits of its static knowledge of the data. For instance, it would ignore the fact that all staff members have names, codes, and ages. Compare this with the code required to perform the same task over the object `staff` equivalent to d':

```
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 with the task's semantics. Its also more succinct: generic operations on staff and staff members can easily be factored out in class definitions, thoroughly tested, and then reused. The programmer can also count on finer-grained system type checking to ensure code correctness: the age has the properties of an integer number, and all members have one and only one name, so the system can detect erroneous manipulations at compile time.

Overall, SAX and DOM do not preserve the semantics of XML encodings of real–world entities. When computations must address, respectively, the syntax and structure of the data, both solutions can be quite effective. For example, SAX offers an efficient and simple way of counting the number of elements or the occurrences of a particular string within the document. Similarly, DOM is ideal when it comes to adding or removing "has–a" relationships between document components.

Large classes of computations, however, are directly concerned with the intended meaning of the data rather than its syntactic or structural properties. In these cases, the data should be represented within the language by values whose semantics reflect the intended meaning as closely as possible. When a binding does not satisfy these expectations, the programmer faces the choice of expressing computations in an awkward algebra or writing ad hoc translation code between the bound values and the preferred ones.

For example, many SAX and DOM programmers would prefer to tackle the proposed task by mapping the string or tree the parser generates into `staff` and then computing directly over `staff`. In a statically typed language, this requires the preliminary declaration of type `Staff` as a description of the semantics intended for d'. This suggests the possibility of defining automated solutions that take types as the input to semantic–preserving bindings to XML data.

JAXB

In the Java Architecture for XML Binding (JAXB), type information is automatically generated from document descriptions, such as DTDs. Specifically, descriptions are converted into classes with unmarshaling functionality, in that they can recursively generate their own instances from valid XML documents.

For example, suppose the fragment d' of d exists as a stand–alone document associated with the following DTD STAFF :

```
<!ELEMENT staff (member*)>
<!ELEMENT member (name)>
<!ATTLIST member code CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
```

The JAXB programmer passes STAFF to a schema compiler, which generates the declarations for two classes `Staff` and `Member`. These classes resemble those shown earlier but also include a static method `unmarshal` for binding to XML encodings

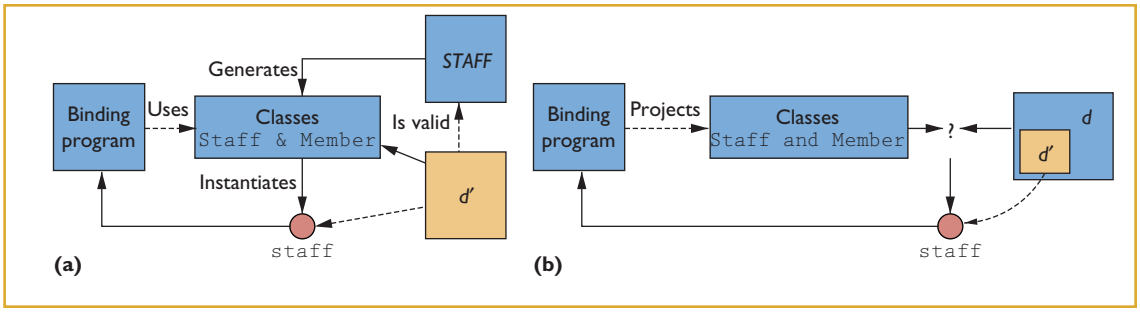

Figure 1. (a) JAXB bindings are driven by document descriptions, while (b) SNAQue bindings are driven by program types.

of their instances.

The programmer may then add domain–specific functionality to the generated classes and use them in programs. To compute over *d′*, applications invoke the `unmarshal` method of `Staff` on *d′*. Assuming that `doc` denotes the file that contains *d′*, the instantiation of *d′* would look like

```
Staff staff = Staff.unmarshal(doc);
```

Thereafter, applications can manipulate `staff` with the host language's standard facilities (see Figure 1).

Binding is successful only if the target document is valid with respect to the initial DTD — otherwise, the marshaling method raises an exception. The schema compiler guarantees this by augmenting the binding logic within the generated classes via programmatic checks equivalent to the constraints expressed in the DTD. Validity is thus enforced at runtime, with a single dynamic check in an otherwise statically typed program.

In addition, the generated classes can marshal their instances back into XML documents, thereby implementing a lightweight, file–based update model. Object marshaling is also validated, in that it only generates documents that conform to the initial DTD.

The JAXB programmer can also feed a binding schema to the compiler to specialize its binding strategy. The binding language gives programmers control over generation of class names, properties, and methods, as well as type conversions, constructor functions, type–safe enumeration classes, and even interfaces. A simple binding schema, for example, could specify that `Staff` objects encapsulate arrays of `Member` objects instead of lists and that their `code` property is an integer rather than a string.

Although JAXB succeeds in preserving data semantics, it has some shortcomings. Some of these are certainly related to its early implementation status (such as the unstable and incomplete functionality, including the limited support for description formalisms). Others occur more noticeably in the form of loss of type information, as JAXB enforces DTD-imposed constraints at the point of marshaling rather than when they are actually violated.

The main issues, however, come from the architecture's dependency on document descriptions. First, programs cannot directly bind to document fragments and thus access regular subsets of semistructured documents; for example, JAXB programs cannot bind directly to *d′* within the more irregular *d*. Instead, they must provide a DTD for the entire *d* and bind to it. Given the irregularities in *d*, the DTD would probably be rather convoluted, thus compiling into classes that offer poor or excessive abstraction over the actual data. Such classes would provide little static knowledge to the system and a cumbersome, inefficient, cast–based algebra to the programmer. Overall, this approach would result in the introduction of semistructured data into a statically typed language, which is a notoriously inconvenient match. Describing staff members, for example, would require either a single class with the totality of required properties or a rather unnatural class hierarchy of `Member`, `MemberWithHomepage`, `MemberWithPhone`, `MemberWithNameAndPhone`, and so on. In both cases, computations would have to use extensively type casts and conditional statements.

Document–level bindings also show poor resilience to changes in the data. Even when a change is irrelevant to existing programs, the classes generated from old document descriptions become invalid and must be regenerated. Besides the problems class regeneration could introduce in loosely coupled systems, class regeneration discourages programmers from adding domain–specific functionality to the generated classes. For example, a method `find` that takes a member's name and returns the corresponding `Member` object should not be directly declared for `Staff` objects

because future class generations would simply make it vanish. Solutions must therefore be based on wrapper classes or extension patterns, which means the programmer must be aware of the binding architecture's internal workings.

Because of automatic class generation, this complexity is in fact propagated throughout the binding process. To control the generation, the programmer must learn a new binding language, which becomes complicated in most nontrivial cases. Programming control remains partial due to the unavoidable tension between the mechanism's generality and the computational requirements' specificity. Finally, the generated classes remain rather opaque to the programmer and can prove difficult to integrate with legacy code.

SNAQue

In the SNAQue binding architecture, document descriptions play no role, and binding programs directly project type information over XML documents.

A type — possibly derived from inspecting the document or, when available, a document description — is then projected by the programmer over the document in an attempt to find a conforming fragment. This entails a recursive match between the structure of the type and the names of the elements in the document. If the match is successful, the system transforms the conforming fragment into an instance of the projected type and returns the latter to the binding program.

SNAQue is thus similar to mechanisms used extensively in statically typed languages to guarantee type–safe access to heterogeneous or persistent data. These include the numerous language incarnations of infinite, untagged union types (for example, Simula-67's subclass structure or Amber's type `dynamic`), especially their implementations in persistent languages (for example, Napier88's type `any`). Instead of performing a dynamic check between the type the computation projects and the type associated with the data, however, SNAQue compares the former directly with the self–description embedded in the document.

The programmer now has full control of the projected types — which might have been defined for binding purposes or simply reused from legacy code — and the design in which they should participate. Bindings can be defined with an arbitrary degree of granularity and thus enable access to fragments of possibly semistructured XML documents. Moreover, irrelevant changes to the data have no effect on projected types or binding programs.

For the time being, the SNAQue architecture focuses on bindings. An update model over the bound data is not precluded, however, and SNAQue could here offer a finer-grained model than JAXB. This could introduce validity issues because language types cannot capture the full range of document constraints but instead require programmatic checks. While JAXB automatically generates these checks from document descriptions, SNAQue leaves such responsibility to binding programs.

A Binding Example

Consider an object–oriented program that wants to bind to *d′* using SNAQue. Now, the program declares the classes `Staff` and `Member` shown earlier and then projects `Staff` on *d*. The binding mechanism infers a structural type from `Staff` and begins a recursive analysis of both type and document in an attempt to match their structures.

Because `Staff` has only an instance variable `member` of class `Member[]`, the mechanism considers only homonymous elements immediately below the `staff` element. In the second step, it repeats the analysis between the type inferred from class `Member` and the subdocuments of *d* rooted in each of the `member` elements identified previously:

```
<member code ="123345">
   <name>Steve Neely</name>
   <ext>4565</ext>
   <project>
      <name>SNAQue</name>
   <project/>
</member>
```

In particular, the mechanism looks in each subdocument for at least a `name` element and a `code` attribute. During the third step, it repeats the analysis comparing the atomic type `String` and `int` with the CDATA and PCDATA content of, respectively, the `name` elements and `code` attributes identified in the previous step. The comparisons are all successful because any PCDATA content is a string, whereas the system can convert the CDATA value of all `code` attributes into an integer.

At this point, the fragment of *d* that conforms to the type inferred from class `Staff` is the one obtained by traversing *d* along the elements and attributes that the recursive analysis successfully considered (for example, *d′*). The last step is then to convert *d′* into the `Staff` object `staff` and return `staff` to the binding program, as shown in

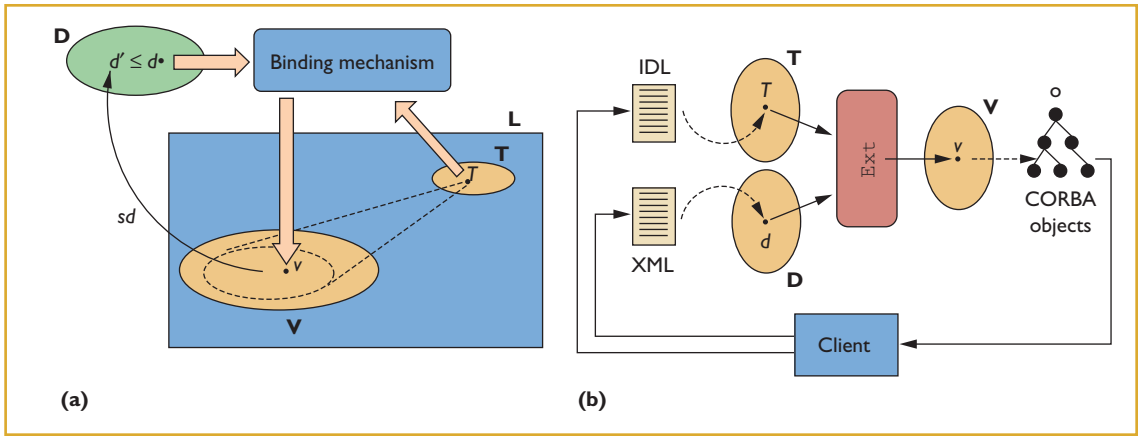

Figure 2. (a) SNAQue and (b) SNAQue/J architectures.

Figure 1b. (You can find more detailed examples of SNAQue bindings at www.cis.strath.ac.uk/research/snaque/examples.)

Formal Definition and Canonical Specification

The SNAQue architecture can be more formally defined as follows. Let D be the set of XML documents, L a statically typed programming language, and V and T the value and type spaces of L, respectively (see Figure 2a). Let sd: V → D be a mapping that gives an interpretation of language values as self–describing documents. Let also ≤ in D × D be a relation of containment between XML documents that gives an interpretation of the notion of document fragments.

Definition 1. Let $v \in$ V. v is extractable from $d \in$ D according to $T \in$ T if v has type T and there exists $d' \leq d$ such that $sd(v) = d'$.

Finally, a SNAQue binding 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.

Elsewhere, we have shown that we could correctly implement the SNAQue architecture, and we did it for a canonical language core L.[3] In doing so, we followed a principle of generality that allows binding mechanisms for other typed languages to be derived directly from L.

The canonical language L is defined around a value notation V, a type language T, and a typing relationship between the two. In particular, we chose a selection of the first–order types commonly found in existing procedural languages for T. We can recursively build canonical types from a set of atomic types and include record, set, union, and recursive types. Specifically, a type $T \in$ T is one of the atomic types $B_1, B_2, ..., B_N$, a record type $[l_1:T_1, ..., l_n:T_n]$, a set type $set(T)$, a union type $T_1 + T_2$, or a recursive type $\mu X.T$, where X is a type variable and the operator μ binds occurrences of X in T.

Canonical values mirror the available types. A value $v \in$ V is an atomic value $b_k \in B_k$, a record value $[l_1 = v_1, ..., l_n = v_n]$, a set value $\{v_1, ..., v_n\}$, or the empty set $\{\}$. The typing relation is inductively defined in a standard fashion. An atomic value b_k has the corresponding type B_k, whereas a record $[l_1 = v_1, ..., l_n = v_n]$ has the type $[l_1:T_1, ..., l_n:T_n]$, but only if each v_i has type T_i. A set $\{v_1, v_2, ..., v_n\}$ has the type $set(T)$ only if all the v_i have type T; the empty set has the type $set(T)$ for all T. A value v thus has type $T_1 + T_2$ if v has type T_1 or type T_2 and, finally, v has type $\mu X.T$ if v has the type obtained by substituting $\mu X.T$ for all the bound occurrences of X in T.

For the canonical mechanism's specification, we have defined a fairly simple self–describing interpretation of canonical values, although the lack of XML support for set values introduces some subtlety in the definition of sd. Therefore, we defined the relation of document containment ≤ by following the intuition that a document is a fragment of another if both are well formed and the first is syntactically contained in the second, as in the case of d' and d.

In the canonical binding mechanism, for example, the binding would require the projection of the following canonical type over d:

```
[member : set([name : String, code : int])]
```

and result in the identification of the following canonical value equivalent to d':

```
[member : {[name = ''RichardConnor'',
```

```
code = 123517],
        [name = ''SteveNeely'', code
                 = 123345],
        [name = ''FabioSimeoni'', code
                 = 175417]}]
```

Our algorithm `Ext` takes an XML document *d* and a canonical type *T* and returns a canonical value *v*, which is extractable from *d* according to *T*. In particular, we have proved the soundness and completeness of `Ext` with respect to the canonical specification.[3]

SNAQue/J

The SNAQue architecture for the Java language (SNAQue/J) consists of an API with a single public class `SNAQueJ` that exposes a static method `bind` for binding to fragments of XML documents. In particular, `bind` takes a `Class` object and a reference to an XML document, and returns an instance of the class corresponding to the `Class` object.

Consider the university department staff example presented earlier and assume that `clStaff` and `doc` denote the `Class` object corresponding to `Staff` and the document *d*, respectively. SNAQue/J is then invoked simply as

```
Staff staff = (Staff)
     SNAQueJ.bind(clStaff,doc);
```

If the binding fails, an exception is raised; otherwise, the program can downcast the returned object to the projected class.

We derived SNAQue/J directly from the canonical mechanism discussed earlier. The projected classes are mapped onto canonical types before invoking `Ext` and then, as Figure 2b shows, the mapping is used to guide the conversion of the canonical values returned by successful bindings into equivalent graphs of Java objects. The same extension scheme can be used to derive implementations of the SNAQue architecture for other typed languages.

SNAQue/J is thus completely defined by the mapping between Java classes and canonical types. The mapping is established at binding time via a reflective analysis of the projected class. Roughly, the mapping behaves like the identity function on atomic types while it associates classes with record types (possibly recursively defined). For example, class `Member` would map onto the record type `[name : string, code : int]`.

We derive union and set types by inferring additional information at binding time. In particular, SNAQue/J maps an array class or any class that implements the `Collection` interface onto a type *set*(*T*), using the names of instance variables to deduce the type *T*. Consider, for example, the following version of class `Member`:

```
class Member {
   String name;
   int code;
   List project;}
```

where the intended class of objects in list `project` is

```
class Project {
      String name;
      Member coordinator;}
```

When `Member` is projected over an XML document, SNAQue/J uses the name `project` of the `List` variable in `Member` to infer the following recursive type:

```
μX.[name : String, code : int, project :
    set([name : String, coordinator : X])]
```

Inferring union types is less trivial because Java — like most object–oriented languages — offers no direct support for them. Partly, this is because much of the union types' flexibility is achieved with inheritance. In particular, we infer nondisjoint union types from class hierarchies and disjoint union types from structurally disjoint classes that implement the same (possible empty) interface. For example, if the following class is in scope when `Member` is projected,

```
class Professor extends Member {
      Project[] supervisedProject;}
```

SNAQue/J maps class `Member` into the following type:

```
μX.[name : String, code : int, project :
     set([name : String, coordinator : X])]
+
μX.[name : String, code : int, project :
    set([name : String, coordinator : X]),
    supervisedProject : set([name : String,
    coordinator : X])]
```

As an example of disjoint union types, consider a binding to a semistructured XML document in which university projects are represented by `project` elements having either a `name` or a `title` subelement. To capture this terminological diversity, the binding program might first declare an interface

Table 1. Characteristics of SAX, DOM, JAXB, and SNAQue.

Header?	Data structure	Computational model	Application domain
SAX	String	Parsing events and callbacks	Syntax–oriented processing
DOM	Tree	Tree navigation	Structure–oriented processing
JAXB	Document dependent (via type generation)	Generic	Semantics–oriented processing mostly in tightly coupled systems (regular and stable data)
SNAQue	Computation dependent (via type projection)	Generic	Semantics–oriented processing mostly in loosely coupled systems (data with semistructured features)

`Project` with a single method `getName()`. It could then declare two classes `Project1` and `Project2` that implement `Project` using a `String` instance variable called `name` and `title`, respectively. Finally, the program might project `Project` over the target document and let SNAQue/J infer the following type:

```
[name : String] + [title : String]
```

Depending on their content, SNAQue/J would convert the project elements in the document into instances of either `Project1` or `Project2`.

Conclusions

In contrast to SAX and DOM, JAXB and SNAQue preserve data semantics and thus promote programming simplicity and safety (see Table 1). Particularly in SNAQue, bindings are driven by partial and user–specified type projections rather than document descriptions, as in JAXB. This makes defining and maintaining them simpler, especially in the presence of semistructured and rapidly evolving data.

The results suggest that type projections may be fruitfully used within a dedicated XML language rather than at its boundary with the file system or network. Early tests of the hypothesis appear elsewhere[4,5] in the context of an object–based and a mixed–paradigm (procedural and query) model, respectively. Interestingly, the tests bring out the similarity between partial projections over untyped but self–describing data and structural record subtyping of statically typed data. Besides achieving flexibility within a safe environment, however, projections appear also to enable partial typing — typed and untyped views of the same data within the same computation. Further work in this direction is nonetheless required.

References

1. C. Low, J. Randell, and M. Wray, *Self–Describing Data Representation (SDR)*, Hewlett Packard Labs, 1997.
2. P. Buneman, "Semistructured Data," *Proc. 16th ACM SIGACT–SIGMOD–SIGART Symp. Principles of Database Systems*, ACM Press, 1997, pp. 117–121.
3. F. Simeoni et al., "An Approach to High–Level Language Bindings to XML," *Special Issues in Information & Software Technology*, Elsevier, 2002, pp. 217–228.
4. R. Connor et al., "Projector: A Partially Typed language for Querying XML," *Plan–X: Programming Language Technologies for XML*, 2002; www.research.avayalabs.com/user/wadler/planx/planx-eproceed/proceed.html..
5. P. Manghi et al., "Hybrid Applications over XML: Integrating the Procedural and Declarative Approaches," submitted for publication, *Proc. WIDM 2002: Web Information and Data Management*, ACM Press, 2002.

Fabio Simeoni is a research fellow in the Computer Science Department at the University of Strathclyde. His research interests include programming language design, programming models over semistructured and self-describing data, and architectures for loosely-coupled distributed systems. Contact him at fabio.simeoni@cis.strath.ac.uk.

David Lievens is pursuing a PhD at the University of Strathclyde. His research interests include programming languages, middleware and distributed systems. He received his BSc in computing Science from the Universiteit Gent, Belgium. Contact him at david.lievens@cis.strath.ac.uk.

Richard Connor is the chair of computer science at the University of Strathclyde. His research interests span persistent object systems, database programming languages, type systems and programming system implementation. Contact him at richard.connor@cis.strath.ac.uk.

Paolo Manghi is a research fellow in the Computer Science Department at the University of Pisa in Italy. His work mainly focuses on programming over XML data, specifically on XML query languages and type systems (TeQuyLa, TQL, and microXQuery projects) and programming language bindings to XML data. Contact him at manghi@unipi.di.it.