A Generative Approach to the Implementation of Language Bindings for the Document Object Model*

Luca Padovani   Claudio Sacerdoti Coen   Stefano Zacchiroli
{lpadovan, sacerdot, zacchiro}@cs.unibo.it

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 – 40127 Bologna, IT

Abstract. The availability of a C implementation of the Document Object Model (DOM) offers the interesting opportunity of generating bindings for different programming languages automatically. Because of the DOM bias towards Java-like languages, a C implementation that fakes objects, inheritance, polymorphism, exceptions and uses reference-counting introduces a gap between the API specification and its actual implementation that the bindings should try to close. In this paper we overview the generative approach in this particular context and apply it for the generation of C++ and OCaml bindings.

1 Introduction

The widespread use of XML imposes every mainstream programming language to be equipped with an implementation of the main W3C technologies, among which is the Document Object Model (DOM) [7, 8]. Quite often these standard technologies require substantial development efforts that small communities cannot always afford. One way to address this problem is the introduction of a common platform, like the recent .NET, that enables direct library sharing among different languages. The alternative, traditional solution is to implement a given API in a low-level programming language (typically C) and to write *bindings* to this implementation for all the other programming languages. This approach has the advantage of requiring no modification to the language implementation. On the other hand the low-level implementation introduces a gap between the original high-level specification of the API and the low-level provided interface. The gap must be closed in the binding process by providing a high-level API that is as close as possible to the original specification. In this paper we overview a framework for the generation of bindings for a C DOM implementation.[1]

* This work was partly supported by the European Project IST-2001-33562 MoWGLI. Luca Padovani received partial support from the Ontario Research Centre for Computer Algebra.

[1] We take `Gdome` [1, 5], the `GNOME` DOM Engine, as our representative DOM implementation. See `http://gdome2.cs.unibo.it`.

The DOM API, as well as several other W3C technologies, is blatantly biased towards an object-oriented programming language with support for inheritance, polymorphism, exceptions and garbage collection, like Java. This bias is perfectly acceptable in the bleak world of mainstream programming languages where even sensibly different languages tend to converge towards a common, limited set of syntactical and semantical aspects. In other communities this tendency turns out to be too constraining. In particular, the larger the gap between the binding target language and Java, the greater the efforts for implementing smoothly the aforementioned APIs.

In order to get a clearer idea of the problems, let us have a look at some code that uses the `Gdome` C DOM implementation. The following code fragment is meant to perform an apparently trivial task: iterate over the child elements of a given element `el`, and perform some unspecified action on each of them:

```
GdomeException exc;
GdomeNode *p = gdome_el_firstChild (el, &exc);
while (p != NULL) {
  GdomeNode *next = gdome_n_nextSibling (p, &exc);
  if (gdome_n_nodeType (p, &exc) == GDOME_ELEMENT_NODE) {
    GdomeElement *pel = gdome_el_cast (p);
    /* do something with the element */
  }
  gdome_n_unref (p, &exc);
  p = next;
}
```

Note how: (1) each method invocation carries an extra argument `exc` for detecting exceptions and that, for brevity, we do not perform error checking; (2) safe downcasting is done via explicit macros, but there is always the dangerous temptation of using C casting because, all considered, "it works"; (3) reference-counted objects requires lots of careful coding to avoid memory leaks. Besides, `Gdome` uses its own UTF-8 strings meaning that, prior to calling any method accepting a string as a parameter, the programmer has to allocate a `GdomeDOMString` object, initialize it, and free it after the call. These details, which are not visible at the DOM specification level, sensibly increase the programmer's discomfort and annoyance.

Since DOM provides a set of standard interfaces with good uniformity of names and types, there is the interesting opportunity of generating language bindings automatically, instead of hand-coding them. The advantages of this approach include reduced development time, increased maintainability and modularity of the bindings. Besides, an XML specification of the DOM interfaces is already available within the DOM recommendation itself. Although its main use is for generating documentation and test cases, it happens to have most of the information needed for the generation of a language binding once the appropriate design choices have been made.

The architecture we propose for the automatic generation of DOM bindings is shown in Figure 1. On the left hand side we have the XML specification of

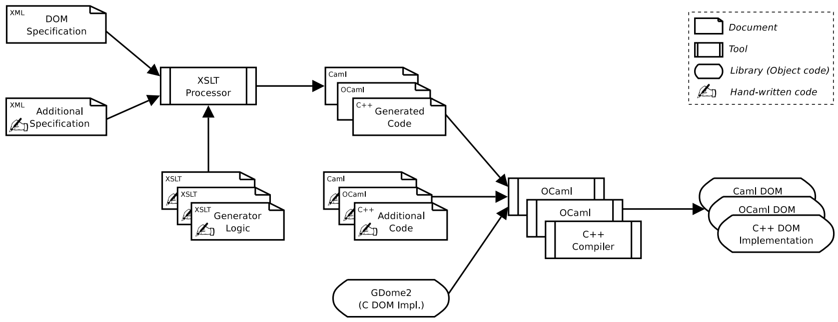

Fig. 1. Architecture of the generator for DOM bindings.

the DOM interfaces. Most of the specification is the actual W3C XML specification, but, as we will see in Section 2, we need additional information that is contained in a separate document. Being the specification encoded in XML, we have implemented the generator as the combination of an XSLT engine along with a set of XSLT stylesheets [6]. The code of different bindings based on different binding logics is generated by different XSLT stylesheets. The generated code is then combined with hand-written code which cannot be generated automatically (it amounts to a handful of lines) and linked against the `Gdome` DOM implementation in order to produce the final binding library.

The outline of the paper is as follows: in Section 2 we overview the format of the high-level DOM specification, what kind of information is available and how it is organized. Sections 3 and 4 cover the main design choices for the bindings currently implemented: C++ and OCaml [12]. In particular, we emphasize those aspects of the DOM API for which C++ and OCaml provide more natural implementations than C does. In Section 5 we show three templates for the three bindings highlighting similarities and differences among them. These also show what kind of XSLT code is required to be written when following our approach. We conclude the paper in Section 6 trying to quantify the effectiveness of the whole approach. The source code of the framework is freely available at the address `http://gmetadom.sourceforge.net`. In the rest of the paper some knowledge of the Document Object Model is assumed.

2 DOM Interface Specification

In this section we briefly overview the XML description of the DOM interfaces as it is provided by the W3C.[2] Each DOM interface is described by an `interface` element characterized by a name and possibly the base interface this interface

[2] This description is not directly advertised by the W3C, it can be found as part of the source XML files of the DOM recommendation, `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.zip`.

derives from. In the following example we see the `Element` interface extending the `Node` interface:

```
<interface name="Element" inherits="Node" id="ID-745549614">
 <descr>...</descr>
 ...
</interface>
```

In several places within the XML elements there are `descr` elements whose content is meant to be informal and used for documentation purposes. Although useless for code generation, they may be used for producing documenting comments aside method and class declarations in the generated code, possibly adhering to some standardized conventions.[3]

The most important components of an interface are the descriptions of attributes and methods. Each attribute is specified in terms of a name, a type, and a `readonly` flag. Here is the declaration for the `nextSibling` attribute in the `Node` interface:

```
<attribute readonly="yes" type="Node" name="nextSibling"
           id="ID-6AC54C2F">
 <descr>
  <p>The node immediately following this node.
   If there is no such node, this returns
   <code>null</code>.</p>
 </descr>
</attribute>
```

Methods are characterized by a name, a list of parameters, the type of the returned object, and an optional `raises` section which lists the exceptions that can be raised by the method. Each parameter comes with a name, a type, and passing style which, in the case of DOM interfaces, is always `in` (that is, input parameters). Here is the declaration for the `replaceChild` method in the `Element` interface:

```
<method name="replaceChild" id="ID-785887307">
 <descr>...</descr>
 <parameters>
  <param name="newChild" type="Node" attr="in">
   <descr>...</descr>
  </param>
  <param name="oldChild" type="Node" attr="in">
   <descr>...</descr>
  </param>
 </parameters>
 <returns type="Node">
  <descr>...</descr>
 </returns>
 <raises>
```

[3] These are typically simplified forms of literal programming.

```
  <exception name="DOMException">
   <descr>
    <p>HIERARCHY_REQUEST_ERR: ...</p>
    <p>WRONG_DOCUMENT_ERR: ...</p>
    ...
   </descr>
  </exception>
 </raises>
</method>
```

There is only one DOM exception (at least within the Core DOM module), which is parameterized by an exception code (whose symbolic names can be seen in the description within the `exception` element).

Occasionally an interface also defines a list of constants within a `group` element. Each constant has a symbolic name, a type, and a value. The most important constants are those determining the type of a DOM node and can be found inside the `Node` interface:

```
<group id="ID-1841493061" name="NodeType">
 <descr>
  <p>An integer indicating which type of node this is.</p>
  ...
 </descr>
 <constant name="ELEMENT_NODE" type="unsigned short" value="1">
  <descr>...</descr>
 </constant>
 <constant name="ATTRIBUTE_NODE" type="unsigned short" value="2">
  <descr>...</descr>
 </constant>
 <constant name="TEXT_NODE" type="unsigned short" value="3">
  <descr>...</descr>
 </constant>
 ...
</group>
```

There is one piece of information that is missing from the XML description of the DOM interfaces and that is important during the code generation phase: no indication is given as to whether an attribute, a parameter or the value returned by a method can be *null*.[4] This detail has no practical consequences in Java simply because in this language `null` is a special, distinguished value that any pointer can take. But this is not the case in general: for example, a C++ reference cannot be null; hence C++ references cannot be used for representing types of nullable entities. Even more delicate are functional languages, where there is usually no notion of "null object", or at least this notion has to be used in a type-safe manner: there can be an empty list or an empty tree, but the general

[4] In some cases this can be heuristically inferred by the description of the entity (see the `nextSibling` attribute shown previously). What we mean is that there is no systematic, exploitable information about this fact.

approach for representing nullable or optional values is to use the α `option` type (in Objective Caml and SML) which permits the construction of "an object x" as `Some` x and the construction of "no object" as `None`. Symmetrically, optional values must be properly deconstructed using pattern matching. Clearly this may become annoying and exceedingly verbose, especially in those cases when a parameter is *required to be non-null* or a returned value *cannot be null*. What is missing is a `nullable` attribute in the XML specification of DOM attributes, methods and parameters indicating whether that particular entity admits `null` among the possible values. This way, during the code generation phase, we are able to generate the most appropriate type. In particular, when generating code for a functional language we are forced to use optional values only where strictly needed. As a side effect, the API produced is also lighter and more usable.

Instead of modifying the original XML specifications, we have preferred to store this information in a parallel set of complementary XML resources, called annotations. A fragment of these annotations relative to the `Node` interface is shown below:

```
<Annotations>
 ...
 <Attribute name="parentNode" nullable="yes"/>
 <Attribute name="childNodes" nullable="no"/>
 ...
 <Method name="insertBefore" nullable="no">
  <Param name="newChild" nullable="no"/>
  <Param name="refChild" nullable="yes"/>
 </Method>
 ...
</Annotations>
```

3 C++ Binding

We have decided to keep the C++ binding as much lightweight as possible, considering the overhead that is intrinsic in the layered architecture of the DOM implementation.[5] The C++ binding consists of a set of classes, one for each DOM interface, which act like wrappers for the corresponding, lower-level C structures. Wrappers provide one method for each method in the DOM interface, and getter/setter methods for attributes (the setter method is only generated if the attribute is writable). The classes are declared in the `GdomeSmartDOM` namespace, which is typically abbreviated in the application source code with an alias directive

```
namespace DOM = GdomeSmartDOM;
```

Basically the wrapping classes are *smart pointers* [9–11] that relieve the programmer from worrying about memory management. The only unusual feature

[5] The `Gdome` DOM implementation is itself a wrapper for a non-conformant, DOM-like library.

is that the counter is already present at the Gdome level, so the wrappers only automate the increment and decrement operations. Each wrapper class has two constructors:

```
class Node
{
public:
  explicit Node(GdomeNode* = 0);
protected:
  explicit Node(GdomeNode* obj, bool) : gdome_obj(obj) { }
  ...
};
```

The public constructor behaves normally from the developer's viewpoint, it increments the reference counter of the Gdome object whenever this is non-null. The protected constructor, distinguished because of an unused boolean argument, is used internally in the generated code for initializing a wrapper with an object returned by a Gdome method: Gdome already increments the counter of any returned object, hence it would be an error to increment it twice (or we would have to decrement it explicitly at some extra cost). The following code fragment shows the generated implementation of the setAttributeNode method in the Element interface:

```
Attr
Element::setAttributeNode(const Attr& newAttr) const
{
  GdomeException exc_ = 0;
  GdomeAttr* res_ =
    gdome_el_setAttributeNode(
      (GdomeElement*) gdome_obj,      // self
      (GdomeAttr*) newAttr.gdome_obj, // projection Attr -> GdomeAttr*
      &exc_);
  if (exc_ != 0)
    throw DOMException(exc_, "Element::setAttributeNode");
  return Attr(res_, true); // promotion GdomeAttr* -> Attr
}
```

As inheritance and polymorphism are handled at the Gdome level, the only support that we have automated has been the provision for casting. This is possible because the XML description of the DOM interfaces includes information about the relationship between interfaces. Upcasting is implemented by deriving wrapping classes for extended interfaces from wrapping classes for the base interface:

```
class Element : public Node { ... };
```

Safe downcasting is implemented by generating, in each wrapping class for a derived interface, a set of constructors taking as a parameter a wrapper class for an ancestor interface:

```
class Text : public CharacterData
{
  // ...
  Text(const Text&);
  Text(const CharacterData&);
  Text(const Node&);
};
```

If the cast fails the wrapper object is initialized with a NULL pointer (this is the same behavior of the dynamic_cast operator on plain C++ pointers). Casts like Element to Text, which are statically known to be unsafe, are prevented simply because no suitable copy-constructor that applies is generated. Finally, appropriate cast constructors are also generated whenever a DOM object is meant to implement multiple DOM interfaces (this is the case of Node and EventTarget, which belong to different, orthogonal DOM modules).

String management had to be coded manually, but since it is completely unrelated to all the other design choices related to the C++ binding it has been separated from the rest of the code so that it is reusable from alternative C++ bindings that we might want to experiment with in the future. Gdome has its own (reference counted) structures for representing strings encoded in UTF-8, whereas C++ comes with an extremely general string implementation in the STL library, but such implementation does not take into account encoding issues directly. We wanted to provide easy conversions from Gdome's internal encoding (which, in principle, the programmer should not be concerned about), to a limited set of generally useful encodings (UTF-16 and UCS4) without introducing at the same time gratuitous inefficiencies. Hence we have structured string management on two levels. At the bottom level is the DOM::GdomeString class which is just a wrapper to Gdome's GdomeDOMString type and provides only basic operations like comparison and initialization from plain C++ strings. At a higher level are DOM::UTF8String, DOM::UTF16String, DOM::UCS4String, which are typedefs for instantiations of std::basic_string with appropriate char types chosen to match the required number of bits but also the standard C++ char types. On most architectures, DOM::UTF8String and DOM::UCS4String are just aliases for std::string and std::wstring respectively.

Appropriate operators and constructors are provided for transparent encoding translation passing through the DOM::GdomeString type. It is thus possible to write

```
  DOM::UTF16String s = n.get_nodeName();
  // ...do something with s...
  n.set_nodeValue(s);
```

without worrying about the fact that those methods return and accept DOM::GdomeString parameters. Note however that a string returned by a DOM method and passed unchanged to another DOM method need not go through any conversion.

Once all the pieces are put together, the conceptually simple but obfuscated example shown in Section 1 can be recast to the following crystal clear piece of C++ code:

```
for (DOM::Node p = el.get_firstChild(); p; p = p.get_nextSibling())
  if (DOM::Element pel = p) {
    // do something with the element
  }
```

4 OCaml Binding

Objective Caml (OCaml) is a class-based, object-oriented extension of Caml, a multi-paradigm language of the ML family whose main characteristics are the strongly-typed functional core augmented with imperative constructs, memory management delegated to a garbage collector, a type-inference based type system that relieves the programmer from having to explicitly declare the type of each function, and an expressive module system that can effectively replace objects and classes whenever late binding is not necessary. Higher-order functions and parametric polymorphism yield a very expressive type system where casts are not necessary at all, and are left out of the language.

The object-oriented capabilities of OCaml differ from traditional object-oriented languages like Java or C++ in that in OCaml subtyping is not related to inheritance. OCaml inheritance is just syntactic sugar to avoid code duplication, and subtyping and interfaces are not necessary: the type of a function or a method can be a generic object type τ that is the type of all the objects that expose *at least* the methods listed in τ. For instance, every object that has a `as_xml` method that returns an XML representation of the object itself matches the generic object type $\tau =$ `< method as_xml: xml ; ... >`, which can be used to type the argument of a method. Despite these differences, a Java class hierarchy can be faithfully mapped using the OCaml class system, so we are able, at least in principle, to provide a DOM binding which can be used in a way that is syntactically similar to more conventional object-oriented languages.

Unfortunately, in OCaml it is not possible to bind directly external functions to object methods. Thus, to provide an object-oriented DOM binding we are forced to adopt a layered approach: the lower level is a binding for the functional core of the language; the upper level builds an object-oriented abstraction on top of the lower level. An user can choose, according to her preferences and programming style, the functional interface of the lower level, or the object-oriented interface. In the latter case the lower level can remain completely hidden.

Assuming the existence of the lower level, the second layer is not complex. A DOM object of type T is implemented as a value of type T' at the lower level and as an object of an OCaml class T'' at the higher level. At the lower level a DOM method is implemented as a pre-method, a function whose first argument is the *self* parameter of type T'. As the C DOM implementation is also based on pre-methods, the OCaml function is easily bound to its native counterpart.

At the higher level the OCaml class has one field of type T' representing *self* at the lower level. This field is hidden to the user, but it is available within the OCaml class via a as_T' method. All the requests sent to an object are delegated to the lower level. Any parameter of type T'' which is pertinent to the higher level is converted to the corresponding type T' at the lower level by means of the as_T' projection. Conversely, any value of type T' returned by the pre-method is wrapped by a new object instance of type T''. The following example, which shows a fragment of the generated implementation of the OCaml `element` class, should clarify the basic mechanisms:

```
class element (self : TElement.t) =   (* TElement.t lower-level type *)
 object
  inherit (node (self :> TNode.t))    (* TElement.t subtype of TNode.t *)
   method as_Element = self           (* projection method *)
   method setAttributeNode ~newAttr =
    let res =
     IElement.setAttributeNode        (* pre-method call *)
      ~this:self
      ~newAttr:
        ((newAttr : attr)#as_Attr)    (* projection attr -> TAttr.t *)
    in
     new attr res                     (* promotion TAttr.t -> attr *)
 end
```

The similarity of the OCaml higher-level binding with the C++ binding should be evident. In both cases, the code builds an object-oriented abstraction on top of a pre-method system. The same similarity is also found between the two binding logics, described in the XSLT stylesheets, and provides further evidence in favor of the generative approach. String management, that was a major design decision for the C++ binding, has also been solved in a similar way for the OCaml binding. Thus the only major difference between the two bindings is the handling of `NULL` values, which was discussed in Section 2.

The low-level OCaml binding presents all the difficulties already faced in the C++ binding and a few new challenges, such as the interaction of the `Gdome` reference counting machinery with the OCaml garbage collector. In particular, the DOM Events Module API requires callback functions for event listeners, thus `Gdome` must register OCaml callback functions. Since OCaml functions are represented at run-time as closures that are managed by the garbage collector, their memory location can change at each collection. As a consequence the generated code is quite involved, and the most part of the bugs found in the development phase were related to memory management.

Notwithstanding the previous considerations, the greatest challenge in the design of the low-level OCaml binding has been related to typing issues. In particular, the problem consists in defining a collection of OCaml types $\mathcal{T}$ for which the subtyping relation is in accordance with the inheritance relation between DOM interfaces. For instance, for the previous example to be statically well-typed, we have to define two types `TElement.t`, `TNode.t` $\in \mathcal{T}$ such that `TElement.t` is a subtype of `TNode.t`.

The simplest solution is declaring a different abstract data type for each `Gdome` class. The type must be abstract since the OCaml compiler has no direct way of manipulating external C values, which are wrapped in garbage collected memory references and which can be manipulated only by external C functions. Since up-casts from a subtype S to a supertype T are always safe C casts in the `Gdome` implementation, we can make functions of the C casts and bound them to OCaml external functions of type $S \rightarrow T$. One major drawback of this solution is that the obtained code is quite inefficient, since the frequently called casting functions, which are nothing more than identity functions, cannot be optimized neither by the C compiler, nor by the OCaml compiler. Moreover, up-casting a value to one of its ancestor classes would require a number of applications of externally defined casting functions equal to the length of the path from the sub-class to the super-class in the inheritance graph or, alternatively, the definition of one distinct casting function between each pair of classes in the inheritance relation.

Due to the previous issues, an alternative, more complex solution has been preferred to the simpler one. The alternative solution is based on a combination of the *phantom types* technique [2, 3] with *polymorphic variants* [4], which are an extension of the type system of OCaml that introduce subtyping in the core language without resorting to the object-oriented extension. As the following explanation should make clear, the combination of the two techniques is cumbersome and definitely not suitable for manual application in large contexts, as a language binding is. Moreover, a local change in the DOM class hierarchy, like those introduced when a new DOM module is implemented, implies global changes in the binding code that are annoying and error prone. Meta-programming solves the problem in an elegant and effective way.

The basic idea is to declare just one abstract phantom data type $\alpha\ \tau$, contravariant in the type parameter α. An instance of $\alpha\ \tau$ is bound to each foreign C type that corresponds to a DOM class. The type $\alpha\ \tau$ is called a *phantom type* since it is parameterized over a type variable α that is not used in the definition of the phantom type and that has no influence on the run-time representation of values. In other words, two values of distinct types $\sigma_1\ \tau$ and $\sigma_2\ \tau$ have the same memory representation and could be casted — if casts were part of the language — to any other type instance $\sigma\ \tau$. The latter propriety is interesting in our context since we want to simulate in the OCaml type system the subsumption rule of the DOM type system, that is casting a value of type $\sigma_1\ \tau$ bound to a DOM class c_1 to the type $\sigma_2\ \tau$ bound to a DOM class c_2 when c_2 is a supertype of c_1. At the same time, we want to statically rule out every pre-method application that is ill-typed at the DOM level. Since OCaml does not have casts nor an implicit subsumption rule, we must simulate both features using parametric polymorphism only.

Our solution consists in declaring any foreign function whose input is a DOM class c_1 and whose output is a DOM class c_2 as having type $[c_1]_i\ \tau \rightarrow [c_2]_o\ \tau$ where $[\_]_i$ and $[\_]_o$ are two encodings from DOM classes to OCaml types such that for each pair of DOM classes c'_1 and c'_2, if c'_1 is a subtype of c'_2 in the DOM type system, then a value of type $[c'_1]_o\ \tau$ is a valid input for a function of type $[c_2]_i\ \tau \to \sigma$ in the OCaml type system. Notice that the latter property is the counterpart for parametric polymorphism of asking $[c'_1]_o\ \tau$ to be a subtype of $[c'_2]_i\ \tau$ for subtype polymorphism. Polymorphic variants [4] can be used to declare two meta-level functions $[\_]_i$ and $[\_]_o$ that satisfy the required constraints.

Table 1. Example of corresponding types in the DOM specification and in the low-level OCaml binding.

	DOM	OCaml
Supertype	Node	[‘Node] τ
Subtype	Element	[‘Node | ‘Element] τ
Relation	Element <: Node because Element extends Node	[‘Node | ‘Element] τ <: [‘Node] τ because α is contravariant in $\alpha\ \tau$ and [‘Node] <: [‘Node | ‘Element] since {‘Node} $\subset$ {‘Node,‘Element}
Method	m : Node $\to T$	m : [> ‘Node | ..] $\tau \to [T]_o$
Object	e : Element	e : [‘Node | ‘Element] τ
Application	m(e) well typed because e : Element $\Rightarrow$ e : Node by subsumption	m(e) well typed because [‘Node | ‘Element] τ matches [> ‘Node | ..] τ

Intuitively, a polymorphic variant value is a tag associated with an optional value (its content). Since we are not interested in the content, we simplify the picture saying that a polymorphic variant is simply a tag. For instance, ‘Node and ‘Element are two values whose tags are respectively Node and Element. A *close polymorphic variant type* lists several distinct tags. For instance, $\sigma_1 =$ [‘EventTarget | ‘Node] is the type of all the values that are either a ‘Node or an ‘EventTarget. The subtyping relation between close polymorphic variant types is in accordance with the subset relation between the sets of tags, hence for example [‘Node] is a subtype of [‘EventTarget | ‘Node]. A *parametric polymorphic variant type* has an unnamed parameter representing an unspecified tag set. For instance, $\sigma_2 =$ [> ‘EventTarget | ..] is the parametric type of all the values that are either an ‘EventTarget or belong to the unnamed tag set parameter indicated by the ellipsis. Moreover, the type $\sigma_1\ \tau$ matches $\sigma_2\ \tau$ (by instantiating the unnamed parameter .. with the singleton set {‘Node}), hence the application of a function of type $\sigma_2\ \tau \to \sigma$ to a value of type $\sigma_1\ \tau$ is well-typed. On the contrary, the application of the same function to a value of type [‘NodeList] τ is not well-typed.

We can interpret the list of tags in a closed contravariant phantom type instance as a list of *provided* capabilities of an object (e.g. providing both the

`EventTarget` and `Node` interfaces in $\sigma_1\ \tau$) and the list of tags in a parametric contravariant phantom type instance as a list of *required* capabilities (e.g. requiring at least the `EventTarget` interface in σ_2). The contravariance requirement must be understood in terms of this interpretation: $\sigma_1\ \tau$ is a subtype of $\sigma_2\ \tau$ when σ_1 has more capabilities than σ_2, i.e. when σ_2 is a subtype of σ_1. Table 1 summarizes the type relationships in the specific case of the `Element` interface.

According to the previous observations, we can implement in XSLT the two meta-level functions $[\_]_i$ and $[\_]_o$ in the following way: let c_n be a DOM class that recursively inherits from $c_1, \ldots, c_{n-1}$ by means of either single or multiple inheritance; we define $[c_n]_i$ as the polymorphic variant type $[> \text{`}C_1 \mid \ldots \mid \text{`}C_n]$ and $[c_n]_o$ as the polymorphic variant type $[\text{`}C_1 \mid \ldots \mid \text{`}C_n]$ where C_i is a tag obtained by mangling the name of the class c_i. For instance, `TElement.t` is defined as $[$`Element`$]_o$ = `` [`EventTarget | `Node | `Element] `` since a DOM `Element` inherits from a DOM `Node` (see the DOM Core Specification [7]) and it is also an `EventTarget` (as described in the DOM Events Specification [8]).

The following example shows the generated code for the low-level OCaml binding of the `setAttributeNode` method:

```
module GdomeT = struct
  type -'a t  (* abstract phantom type, contravariant in 'a *)
 end

module TElement = struct
  type t = [`EventTarget | `Node | `Element] GdomeT.t
 end

module TAttr = struct
  type t = [`EventTarget | `Node | `Attr] GdomeT.t
 end

external setAttributeNode :
 this:[> `Element] GdomeT.t ->
  newAttr:[> `Attr] GdomeT.t ->
   TAttr.t
= "ml_gdome_el_setAttributeNode"
```

The C function `ml_gdome_el_setAttributeNode`, which is also automatically generated as part of the low-level binding, is defined as follows:

```
value
ml_gdome_el_setAttributeNode(value self, value p_newAttr)
{
  CAMLparam2(self, p_newAttr); /* Directive to the garbage collector */
  GdomeException exc_;
  GdomeAttr* res_;
  res_ = gdome_el_setAttributeNode(Element_val(self),
                 Attr_val(p_newAttr), &exc_);
  if (exc_ != 0)
     /* Raises an Ocaml exception */
```

```
    throw_exception(exc_, "Element.setAttributeNode");
  g_assert(res_ != NULL);
  CAMLreturn(Val_Attr(res_)); /* Directive to the garbage collector */
}
```

Once all the pieces are put together, the conceptually simple but obfuscated example shown in Section 1 can be recast to the following pieces of OCaml code:

```
(* Object-oriented *)
let rec iter =
 function
   None -> ()
 | Some p when p#get_nodeType =
    GdomeNodeTypeT.ELEMENT_NODE
   ->
    let p' =
     Gdome.element_of_node p in
      (* do something with p' *)
     iter p#get_nextSibling
 | Some p ->
    iter p#get_nextSibling
in
 iter el#get_firstChild
```

```
(* Purely functional *)
let rec iter =
 function
   None -> ()
 | Some p when INode.get_nodeType p =
    GdomeNodeTypeT.ELEMENT_NODE
   ->
    let p' =
     IElement.of_Node p in
    (* do something with p' *)
    iter (INode.get_nextSibling ~this:p)
 | Some p ->
    iter (INode.get_nextSibling ~this:p)
in
 iter (INode.get_firstChild ~this:el)
```

The code is slightly more verbose than the C++ equivalent. This is mainly a consequence of the bias of the DOM API towards an imperative, first order approach. Note also that, whereas in the object-oriented API inheritance relieves the user from remembering in which class a method is defined, in the purely functional approach each pre-method is qualified with the name of the module it is defined in. For instance, we have to remember that the `get_nextSibling` is a pre-method of the DOM `Node` class.

5 Generator Logic

In this section we look at significant fragments of the XSLT stylesheets that implement the generator logic for the various bindings. We classify the generated code according to the following tasks:

1. *prepare* the pre-method arguments: objects are projected, primitive values are converted if needed;
2. *invoke* the pre-method (high-level OCaml binding) or the `Gdome` method (C++ and low-level OCaml binding);
3. *check* if the pre-method has raised a DOM exception, and, if so, propagate the exception at the target language level. This is only done for the C++ and low-level OCaml bindings only since the high-level OCaml binding uses the same exceptions as the low-level one;
4. *return* the method result by promotion or conversion as necessary.

In addition, each template may also perform a few operations that are specific to the target language.

Here is the template for the C++ binding:

```
1  <xsl:template match="method">
2    <xsl:param name="interface" select="''"/>
3    <xsl:param name="prefix" select="''"/>
4    <xsl:call-template name="returnTypeOfType">
5      <xsl:with-param name="type" select="returns/@type"/>
6    </xsl:call-template>
7    <xsl:text> </xsl:text>
8    <xsl:value-of select="$interface"/>
9    <xsl:text>::</xsl:text>
10   <xsl:value-of select="@name"/>
11     (<xsl:apply-templates select="parameters"/>) const
12   {
13     GdomeException exc_ = 0;
14     <xsl:apply-templates select="parameters" mode="convert"/>
15     <xsl:call-template name="gdome-result">
16       <xsl:with-param name="type" select="returns/@type"/>
17       <xsl:with-param name="init">
18         <xsl:text>gdome_</xsl:text>
19         <xsl:value-of select="$prefix"/>
20         <xsl:text>_</xsl:text>
21         <xsl:value-of select="@name"/>
22         ((Gdome<xsl:value-of select="$interface"/>*) gdome_obj,
23          <xsl:apply-templates select="parameters" mode="pass"/>
24          &exc_)
25       </xsl:with-param>
26     </xsl:call-template>;
27     <xsl:apply-templates select="parameters" mode="free"/>
28     if (exc_ != 0)
29       throw DOMException(exc_,
30       "<xsl:value-of select="$interface"/>
31       <xsl:text>::</xsl:text>
32       <xsl:value-of select="@name"/>");
33     <xsl:call-template name="return-result">
34       <xsl:with-param name="type" select="returns/@type"/>
35     </xsl:call-template>
36   }
37 </xsl:template>
```

The template invocation on line 23 generates the projection code while lines 17–25 generate the `Gdome` method call. The generated code is passed as a parameter to the template `gdome-result` (lines 15–26), which is responsible for adding the promotion code if required. Lines 28–32 produce the code that checks the result of the function, possibly raising a C++ exception. All the residual lines handle the language specific issues (e.g. allocating and releasing temporary variables).

Here is the template for the high-level OCaml binding:

```
1  <xsl:template match="method">
2    <xsl:param name="interface" select="''"/>
3    <xsl:param name="prefix" select="''"/>
4    <xsl:variable name="name" select="@name"/>
5    <xsl:text>  method </xsl:text>
6    <xsl:value-of select="@name"/>
7    <xsl:apply-templates mode="left" select="parameters">
8     <xsl:with-param name="@name"/>
9    </xsl:apply-templates>
10   <xsl:text> = </xsl:text>
11   <xsl:call-template name="call_pre_method">
12    <xsl:with-param name="type" select="returns/@type"/>
13    <xsl:with-param name="isNullable"
14      select="document($annotations)/Annotations/Method
15              [@name=$name]/@nullable='yes'"/>
16    <xsl:with-param name="action">
17     <xsl:text>I</xsl:text>
18     <xsl:value-of select="$interface"/>.
19     <xsl:value-of select="@name"/>
20     <xsl:text> ~this:obj </xsl:text>
21     <xsl:apply-templates mode="right" select="parameters">
22      <xsl:with-param name="name" select="@name"/>
23     </xsl:apply-templates>
24    </xsl:with-param>
25   </xsl:call-template>
26 </xsl:template>
```

The template invocation on lines 21–23 generates the projection code and lines 16–24 generate the pre-method call. The generated code is passed as a parameter to the template `call_pre_method`, which is responsible for adding the promotion code if required. Lines 13–15 retrieve the annotation for the method which contains information about which arguments of the method are nullable.

Finally, here is the template for the low-level OCaml binding:

```
<xsl:template match="method">
  <xsl:param name="interface" select="''"/>
  <xsl:param name="prefix" select="''"/>
value
  <xsl:text>ml_gdome_</xsl:text><xsl:value-of select="$prefix"/>
  <xsl:text>_</xsl:text><xsl:value-of select="@name"/>
  <xsl:text>(value self</xsl:text>
  <xsl:apply-templates select="parameters"/>)
{
  <xsl:apply-templates select="parameters" mode="declare"/>
  GdomeException exc_ = 0;
  <xsl:if test="returns/@type != 'void'">
    <xsl:call-template name="gdomeTypeOfType">
      <xsl:with-param name="type" select="returns/@type"/>
    </xsl:call-template> res_;
  </xsl:if>
```

```
17   <xsl:apply-templates select="parameters" mode="convert">
18     <xsl:with-param name="methodName" select="@name"/>
19   </xsl:apply-templates>
20   <xsl:if test="returns/@type = 'DOMString'">   value res__;</xsl:if>
21   <xsl:if test="returns/@type != 'void'">res_ = </xsl:if>
22   <xsl:text>gdome_</xsl:text>
23   <xsl:value-of select="$prefix"/>_<xsl:value-of select="@name"/>
24   <xsl:text>(</xsl:text>
25   <xsl:value-of select="$interface"/>_val(self),
26   <xsl:apply-templates select="parameters" mode="pass">
27     <xsl:with-param name="methodName" select="@name"/>
28   </xsl:apply-templates>&amp;exc_);
29   <xsl:apply-templates select="parameters" mode="free">
30     <xsl:with-param name="methodName" select="@name"/>
31   </xsl:apply-templates>
32   <xsl:text>if (exc_ != 0) throw_exception(exc_, "</xsl:text>
33   <xsl:value-of select="$interface"/>.<xsl:value-of select="@name"/>");
34   <xsl:call-template name="methodReturn">
35     <xsl:with-param name="name" select="@name"/>
36     <xsl:with-param name="type" select="returns/@type"/>
37   </xsl:call-template>
38 }
39 </xsl:template>
```

Lines 26–28 generate the projection code. Lines 21–28 generate the call to the `Gdome` method and store the result in the local variable `res_`, which is implicitly used later in the `methodReturn` template for the generation of the promotion code (lines 34–37). Lines 32–33 produce the code that checks the result of the function, possibly raising an OCaml exception using the `throw_exception` function.

6 Concluding Remarks

Automatic generation of stubs and bindings is a well-established technique with clear and well-known advantages of increased code correctness and maintenance. Thus the benefits of its application to bindings of W3C APIs — DOM in primis — are not under discussion. In this paper we stress the idea a bit further by developing a framework for the generation of DOM bindings. The main idea of the framework is to exploit the already existent XML specification that, although meant to be just a source for automatic documentation generation, is rich enough for automatic code generation. The missing information that is not available in the XML specification is manually provided and stored in additional XML documents. The binding logic for each target programming language is described in an XSLT stylesheet. Any off-the-shelf XSLT engine can be used as a meta-generator that must be applied to both the binding logic and the XML specification.

To test our framework, we applied it to the generation of a C++ binding for the DOM Core module. Later on we also developed two layered OCaml bindings,

Table 2. Differences in binding logic sizes in number of lines.

	DOM Core module			+ DOM Events module		
	Binding logic (XSLT)	Hand written code (target language)	Generated code (target language)	Binding logic (XSLT)	Hand written code (target language)	Generated code (target language)
C++	768	1405	4289	+40	+74	+915
+ Caml	+1514	+1305	4557	+15	+176	+807
+ OCaml	+640	+291	+2407	+4	+44	+284

one that exposes a functional interface based on pre-methods and another one that exposes an object-oriented interface that is much closer to the DOM API specification. A few months later an implementation of the DOM Events module was also released, and the three binding logics have been updated to cover the new module as well. We are now able to estimate the amount of code that must be provided to add new DOM modules and new binding logics. Table 2 shows the number of lines of code required for the original C++ binding of the DOM Core module, and the number of lines of additional code written to enable the two OCaml bindings and the DOM Events module. In particular, the numbers in the column "+ DOM Events module" show the additional lines related to the Events module with respect to the Core module (i.e. each cell in the right column shows the increment with respect to the corresponding cell in the left column, on the same row). In the first column, the "+ Caml" line shows the increment with respect to the C++ binding, and the "+ OCaml" line shows the increment with respect to the Caml binding (i.e. the object-oriented layer vs the purely functional layer).

The data shown in the table deserve an explanation. First of all, let us consider the case of the addition of a new DOM module to the binding. In any one of the three cases (C++, Caml, OCaml), the number of hand-written lines of code necessary for the new module is very limited, and much smaller than the number of lines necessary for the generation of the first module considered. The number of lines of binding logic that are necessary for the new module is also very small (e.g. 40 lines vs 768 lines for the C++ binding). Nevertheless, the reader could have expected this number to be 0. The reason for having to extend the binding logic can easily be explained: whereas in the DOM Core module there is just one class hierarchy based on single inheritance, the DOM Events module introduces new interfaces that must be implemented by the objects described in the DOM Core module. Thus, to deal with the new module, we had to add multiple inheritance to the binding logic. From the previous observations we can conclude that our framework behaves as expected for vertical extension, that is the application of the framework to new XML specifications.

Let us consider now the extension to a new target language. Comparing the cells of the first column, we note that the effort required for the Caml binding is similar to the effort spent for the C++ binding. On the contrary, the addition of the OCaml binding on top of the previous two bindings was less expensive.

These results can be easily explained. Every code generator is made of a frontend, which interprets the specification, and a backend for the generation of code. The C++ binding and the Caml binding share the frontend, but not the backend. On the contrary, the two OCaml bindings can also share part of the backend, since they have in common several utility functions used by the binding logic, such as name mangling functions or the generator of phantom type instances. One important datum that is not shown in the table is the real amount of code reuse between the C++ binding and the Caml binding, i.e. the number of lines of code that form the frontend. Thanks to the adoption of XML technologies in our framework, this number is basically 0. Indeed, being the specification written in XML, the meta-generator — an XSLT processor — works directly on the parsed XML document, reducing the backend to a very few auxiliary functions and a small set of XSLT template guards. From the previous observations we can conclude that our framework also behaves positively for horizontal extension, that is the application of the framework to new target languages.

Our approach was made practical largely because of the availability of the DOM specification in XML format. Thus we expect that it should be possible to apply our framework without any major modifications to other W3C standards as well, provided that a compliant low-level C implementation exists. Indeed, it is a real pity for the XML specification to be undocumented and almost neglected by the W3C itself. Surprisingly, we have been able to exploit the specification for code generation, but we have faced serious problems when trying to automatically document the generated code. The prose in the XML specification contains pieces of code that are marked to be rendered in a particular way. Nevertheless, the markup elements do not convey enough information to understand the nature of the values in the code. Thus it is often impossible to convert them to the sensible values in the target language. For instance, it is not possible to know whether a value `v` is of a nullable type, to show it in the OCaml examples as `(Some v)`.

References

1. Paolo Casarini, Luca Padovani, "The Gnome DOM Engine", in Markup Languages: Theory & Practice, Vol. 3, Issue 2, pp. 173–190, ISSN 1099-6621, MIT Press, April 2002.
2. M. Fluet, R. Pucella, "Phantom Types and Subtyping". Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002), pp. 448-460, August 2002.
3. Sigbjorn Finne, Daan Leijen, Erik Meijer and Simon L. Peyton Jones, "Calling Hell From Heaven and Heaven From Hell", in Proceedings of the International Conference on Functional Programming, 114-125, 1999.
4. Jacques Garrigue, "Programming with polymorphic variants". In ML Workshop, September 1998.
   `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/variants.ps.gz`
5. Raph Levien, "Design considerations for a Gnome DOM", informal note, 28 March 1999, `http://www.levien.com/gnome/dom-design.html`

6. James Clark (Eds), “XML Transformations (XSLT) Version 1.0”, W3C Recommendation (1999), http://www.w3.org/TR/1999/REC-xslt-19991116
7. Arnaud Le Hors and Philippe Le Hégaret and Gavin Nicol and Jonathan Robie and Mike Champion et al. (Eds), “Document Object Model (DOM) Level 2 Core Specification”, Version 1.0, W3C Recommendation, November 2000, http://www.w3.org/TR/DOM-Level-2-Core/
8. Tom Pixley, “Document Object Model (DOM) Level 2 Events Specification”, Version 1.0, W3C Recommendation, November 2000, http://www.w3.org/TR/DOM-Level-2-Events
9. Daniel R. Edelson, “Smart Pointers: They’re Smart, But They’re Not Pointers”, Proceedings of the C++ Conference, pp. 1–19, 1992.
10. D.Vandevoorde, N. M. Josuttis, “C++ Templates: The Complete Guide”, Addison-Wesley, 2002.
11. Bjarne Stroustrup, “The C++ Programming Language”, Third Edition, Addison-Wesley, 1997.
12. Xavier Leroy and Damien Doligez and Jacques Garrigue and Didier Rémy and Jérôme Vouillon, “The Objective Caml system release 3.07 Documentation and user’s manual”, http://caml.inria.fr/ocaml/htmlman/