The Gnome DOM Engine

Paolo Casarini
Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7
40127
Bologna, Italy
email casarini@cs.unibo.it
web http://www.cs.unibo.it/~casarini

Luca Padovani
Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7
40127
Bologna, Italy
email luca.padovani@cs.unibo.it
web http://www.cs.unibo.it/~lpadovan

ABSTRACT

The widespread use of Web technologies and, in particular, the ever growing number of applications adopting XML [XML00] as the standard language for the encoding of any piece of structured information, naturally calls for efficient implementations of DOM, the standard interface to access the internal structure of documents.

The DOM level 2 API [DOM], which has been conceived as a suitable hierarchy of classes, has its most natural mapping in object-oriented languages such as C++ [CPP] and Java [Java]. This is also testified by the already existing implementations in those languages. However, as of today, most applications are commonly developed in C, because of its standardization, flexibility, efficiency and availability.

In this paper we describe the current state of Gdome2, which provides a DOM implementation for the C programming language [C]. The library is meant to become a key module of the Gnome architecture, supplying a range of facilities for an efficient, portable, and easy management of XML documents in the Gnome way.

We conclude with a comparison between Gdome2 and Xerces, one of the more advanced and actively developed DOM implementations.

INTRODUCTION

Quoting from the W3C (World Wide Web Consortium) page about DOM (Document Object Model) [DOM]

The Document Object Model (DOM) is a platform and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

More concretely, the DOM Level 2 specification defines a hierarchy of interfaces providing an object-oriented API (Application Programming Interface) to navigate and operate on valid HTML (HyperText Markup Language) or well-formed XML (Extensible Markup Language) documents [HUN00] [XML00].

Interfaces are provided for the management of elements, attributes, text nodes, as well as the other XML node types. When speaking of these entities in general, we will call them *document nodes*. In fact, Node is one of the fundamental DOM interfaces from which most of the other entities derive. Basic node operations include creation, deletion and retrieval. Beside that, the DOM specification provides also a set of interfaces for the implementation of event-driven applications. Events can be triggered by physical user actions (such as GUI (Graphical User Interface) events) and by modifications of the document structure and content.

A DOM implementation (also called a host implementation) is a piece of software which implements all (or a subset of) the interfaces defined in the DOM specification. In the simplest view, it takes a parsed XML document and makes it available for processing via some language-dependent implementation of the DOM interfaces. But another fundamental aspect of the whole DOM architecture is that DOM provides just a minimal interface, and that each application can implement more specific classes derived from the basic ones.

The Gnome (GNU Network Object Model Environment) DOM Engine (Gdome2 (Gnome DOM Engine) for short, see http://www.cs.unibo.it/~casarini/gdome2/) is a DOM implementation whose aim is to provide an interface to XML documents to Gnome programmers which is:

• efficient;
• easy to use;
• compliant to W3C standards.

Basically, the problem is that in usual DOM implementations there is a considerable overhead: application are forced to use a fairly narrow interface to be able to modify and access their state. Furthermore, there are also bloated memory requirements for the storing of the state itself.

So, our primary goal is to make the DOM an attractive interface for Gnome[1] applications, in order to save their state, configuration, data and so on. Furthermore, as already pointed out in [LEVDOM99], we are also exploring the pos-

sibility of providing a unified framework allowing mixing of components within the same application, each component being responsible for managing a particular subset of markup inside a XML document (typically, markup belonging to a given namespace [Names99]). This is the so called *document centric* architecture for applications, where the central structure is given by the document and components are plugged in as needed.

Gdome2, whose aim is to address these issues, currently supports "Core" and "XML" modules as defined in the DOM Level 2 Specification[2] but it is supposed to become a full implementation of all the DOM Level 2 interfaces. However, much of the design issues are decided in this phase, because most of the other modules and levels are built on top of this core interface.

REFERENCE TO THE LIBRARY

In this section we will give an overview to some of the APIs implemented in Gdome2, in particular to the non-standard ones. We will also give the skeleton of a simple program based on Gdome2.

API Conventions

In Gdome2, attributes and methods described in the DOM level 2 interfaces are implemented as C functions. In the rest of the paper, we will often use the term *method* to denote the C function implementing the DOM method.

Naming Conventions

Each DOM interface has a corresponding C type whose name is the name of the DOM interface itself with a prefix Gdome. For example, GdomeNode is the type corresponding to the DOM interface Node.

All API function names start with the string gdome_ (identifying the Gdome2 library namespace) followed by an abbreviated prefix indicating the DOM interface the method refers to. In the following table we give the complete correspondence between the DOM Interfaces and the Gdome2 prefixes.

Gdome2 Prefixes

DOM Interface	Gdome2 Prefix
DOMImplementation	di_
DocumentFragment	df_
Document	doc_
Node	n_
NodeList	nl_
NamedNodeMap	nnm_

2. The 2 in the Gdome2 has nothing to do with the Level 2 in the DOM Specification. It is just to distinguish the package from a previous version never released officially.

1. http://www.gnome.org

CharacterData	cd_
Attr	a_
Element	el_
Text	t_
Comment	c_
CDATASection	cds_
DocumentType	dt_
Notation	not_
Entity	ent_
EntityReference	er_
ProcessingInstruction	pi_

Then, the syntax changes depending on the kind of method or attribute access implemented by the function. Explicit methods are provided for reading and writing attribute values (write access to read-only attributes is simply prevented by the lack of the corresponding writing method).

The syntax of Gdome2 API functions is summarized in a more formal below, where *DOMMethodName* and *DOMAttrName* are place-holders for the name of, respectively, interface methods and interface attributes in the DOM specification.

```
Gdome2Method  ::= 'gdome_' InterfacePrefix DOMMethodName
Gdome2SetAttr ::= 'gdome_' InterfacePrefix 'set_' DOMAttrName
Gdome2GetAttr ::= 'gdome_' InterfacePrefix DOMAttrName
```

Parameters Conventions

All Gdome2 functions implementing a DOM API have two extra parameters in comparison with the DOM specification: the first parameter is always a reference to the object for which we are invoking the method.[3] This parameter is usually implicit in object-oriented languages where it is referred to as self or this. The last parameter is always a reference to a GdomeException structure used to store the exception possibly raised by the method invocation (we recall that the C programming language [C] does not have a native mechanism for handling exceptions). All the other parameters in intermediate positions are in a one-to-one correspondence with the parameters specified in the DOM interface.

As an example, the method appendChild of the Node interface is implemented in Gdome2 by a function with the following prototype:

```
GdomeNode *gdome_n_appendChild (GdomeNode *self,
                                GdomeNode *newChild,
                                GdomeException *exc)
```

GdomeDOMString

The implementation of DOMString deserves a section on its own, since it is the only one which differs in some aspects

from the DOM specification. In fact, the specification requires a DOMString to be encoded using UTF (UCS Transformation Format)-16 [Unicode], while Gdome2 invariably uses a UTF-8 encoding [UTF-8]. This is because Gdome2 acts like a wrapper for another library for parsing XML documents, libxml2, which already adopts internally the UTF-8 encoding.

Some of the rationales for this choice are the following (see [encoding] for a more exhaustive treatment of this subject in the context of libxml2):

- UTF-8, while a bit more complex to convert from/to with respect to UTF-16, is also far more compact for a majority of the documents.
- UTF-8 is being used as the de-facto internal standard encoding in the upcoming Gnome text widget [LEVTXT99] [LEVPA99], and a lot of other Unix/Linux code.

In practice, the GdomeDOMString is a simple structure made of a field str which is a pointer to a sequence of bytes assembled as a UTF-8 valid string. So Gdome2 users only need to make sure that characters outside the plain ASCII (American Standard Code for Information Interchange) set are properly converted to UTF-8.

In the near future, we plan to add access and conversion methods taking into account other encodings such as UTF-16.

To manage GdomeDOMString object allocation, Gdome2 has 3 different constructors, depending on the storage allocation class of the source string used as initializer:

```
GdomeDOMString *gdome_str_mkref(const gchar* str)
GdomeDOMString *gdome_str_mkref_own(gchar* str)
GdomeDOMString *gdome_str_mkref_dup(const gchar* str)
```

gdome_str_mkref creates a GdomeDOMString from a statically allocated string. gdome_str_mkref_own must be used when the user wants to make a GdomeDOMString from a dynamically allocated gchar buffer. The buffer will be freed automatically upon destruction of the object. Finally, gdome_str_mkref_dup is similar to the previous one, but a copy of the initializing string is done before construction.

In order to release a GdomeDOMString, just use its destructor gdome_str_unref.

Gdome2 Bootstrap

The first step for any application using Gdome2 is the instantiation of a DOMImplementation object, which provides a number of methods for performing operations independent of any particular instance of the document object model, such as the creation of new Document and DocumentType objects.

The Gdome2 API to create a GdomeDOMImplementation is gdome_di_mkref.

---

3. In fact, this is a quite common implementation choice for object-oriented programming languages.

Standard way to Create a `GdomeDocument`

Within the DOM specification, the only way to create a Document object is by means of the **createDocument** method in the `DOMImplementation` interface.

In Gdome2 this feature is implemented by

```
GdomeDocument*
gdome_di_createDocument (GdomeDOMImplementation *self,
                         GdomeDOMString *namespaceURI,
                         GdomeDOMString *qualifiedName,
                         GdomeDocumentType *doctype,
                         GdomeException *exc)
```

which creates a `Document` object of the specified type with a root element specified by `namespaceURI` and `qualifiedName`.

Gdome2 also implements a non standard way to create a new Document object parsing a XML document identified by a URI (Uniform Resource Identifier):

```
GdomeDocument*
gdome_di_parseFile (GdomeDOMImplementation *self,
                    const gchar* uri,
                    GdomeException* exc)
```

The document can also be validated using `gdome_DOMimplementation_validateFile` which has the same prototype as the method above.

A Simple Example

To illustrate some of the most common functionalities of Gdome2 we present here a simple example solving a frequent task, namely the removal of comments and blank nodes from a DOM tree. This can be useful, for example, to enforce some invariants on the position of nodes inside a validated document.

```
int main (int argc, char **argv)
{
  GdomeDOMImplementation *domimpl;
  GdomeDocument *domdoc;
  GdomeElement *rootel;
  GdomeException exc;

  domimpl = gdome_di_mkref();
  domdoc = gdome_di_parseFile (domimpl, argv[1], &exc);
  rootel = gdome_doc_documentElement (domdoc, &exc);

  cleanSubTree ((GdomeNode *)rootel);

  gdome_di_saveFile (domimpl, "out.xml", domdoc, &exc);

  gdome_di_freeDoc (domimpl, domdoc, &exc);

  return 0;
```

```
}
```

First of all, the Gdome2 user has to bootstrap the library and load the XML data file in a `GdomeDocument` object. Then, he recovers the reference to the root **GdomeElement** of the document by means of the **gdome_doc_documentElement** method.

At that point, he invokes the cleaning function **cleanSubTree** and finally he saves the resulting document back to the file system.

Let us now turn our attention to the cleaning function, described below:

```
void cleanSubTree (GdomeNode *node)
{
  GdomeNodeList *nl;
  GdomeException exc;
  GdomeNode *child;
  long i, nllength;

  nl = gdome_n_childNodes (node, &exc);
  if ((nllength = gdome_nl_length (nl, &exc)) == 0)
    return;

  for (i = nllength - 1 ; i >= 0; i--) {
    child = gdome_nl_item(nl, i, &exc);

    if (gdome_n_nodeType (child, &exc) == GDOME_COMMENT_NODE
                                                        ||
        (gdome_n_nodeType (child, &exc) == GDOME_TEXT_NODE
                                                    &&
          isSpaceStr (gdome_t_data ((GdomeText *)child,
                                                &exc))))
      gdome_n_removeChild (node, child, &exc);
    else if (gdome_n_hasChildNodes (child, &exc))
      cleanSubTree (child);
  }
}
```

First of all, we retrieve a `NodeList` structure with the list of the children of the current node. Next, we have to iterate on each node, possibly removing it if it turns out to be a comment or a discardable space node.

Now, we want to exploit the liveness property of the `NodeList` structure where changes to the tree are automatically reflected in the list. However, this is a problem if one is going to access the list by means of a progressive index, because as the nodes are removed the list becomes shorter and the set of valid indexes smaller. For this reason, we develop the function as a loop where the index decreases at each iteration. In this way, we always access the right node, whether or not the previous one has been removed.

The remaining, important children are recursively visited as they are met.

DESIGN ISSUES

Object-Orientation

One of the first issues to be addressed when considering the implementation of DOM in an imperative language like C is the mapping of the object-oriented architecture of DOM.

Gdome2 objects are implemented in a straightforward way: objects have fields classified into *instance* fields and *class* fields. Instance fields are allocated for each instance of a given class (where the "class" is the realization of a DOM interface), and they are grouped in a standard C structure. Class fields are fields shared by all the instances of a given interface. Each Gdome2 object has one instance field which is a pointer to a statically allocated structure containing its class fields.

A typical example of a class field which is present in all Gdome2 objects is the *virtual table*, containing pointers to the methods implemented by the class of the object itself.

The importance of virtual tables is twofold: a first, straightforward reason is that it allows a mechanism equivalent to *inheritance* of methods, thus favoring the reuse of code and ultimately making easier the maintenance of the source code. But, more important, virtual tables are needed in order to have *late-binding*, which is a common feature of object-oriented languages allowing to delay the choice of the method to be called on an object until the run-time type of the object is known.

Architecture

Although the DOM Architecture can be used for the creation of new documents not residing in some external resource, a DOM implementation is usually not a piece of stand-alone software: in most cases, it is added as a separation layer between a XML parser and user applications.

The distinction between the parser and the DOM implementation is not always so definite. For example, a DOM implementation can provide its own parsing module, so that the DOM layer appears as the lowest one from the application point of view. Alternatively, the XML parser could only be responsible to generate a proper sequence of SAX (Simple API for XML) events, and then it is up to the DOM implementation to create the internal document representation.

As we will see, the approach used in Gdome2 is somehow an hybrid of the two scenarios previously depicted. In fact, Gdome2 is based on the XML C library for Gnome (libxml2[4] for short) which is much more than a mere XML parser. With respect to parsing, libxml2 exports three kinds of interfaces:

- a SAX event-based interface;
- a Pull method, which parses a *whole* document and then returns a corresponding document structure;
- a Push method, where the document is parsed *on demand*, as the application requires for more chunks to be

parsed. This method is conceived to be used, for example, in interactive applications where it is not feasible to block the whole process while parsing a large document.

In particular, the last two methods build a libxml2-dependent tree representation of the document which largely borrows from the DOM core specification, and this is a crucial aspect of the whole Gdome2 implementation, as we will see.

The DOM Tree

In the DOM view, documents have a logical structure which is very much like a tree; to be more precise, which is like a "forest" or "grove", since we can work with several document trees at the same time. The DOM specification is largely organized around the Node interface, which is the basic node that trees are made of and which defines the basic methods for the traversal of the document (such as for retrieving the parent of a node, its first child, the next sibling and so on). More formally, the DOM tree is made of entities that implements the Node Interface.

Internal DOM Tree Representation in Gdome2

As we have seen, libxml2 has its own concept of document tree. So, from one side libxml2 is an already working DOM-like implementation which is relatively light-weight and further justified by the fact that it allows document validation. From the other side, it is definitely *not* a DOM implementation, for it does not export a full DOM interface and it is somehow targeted to the libxml2 internal use of the document.

Thus, Gdome2 acts like a wrapper for the libxml2 tree structure, implementing in a uniform way the DOM interfaces and ultimately hiding any detail relevant to the libxml2 internals. A Gdome2 node is just a wrapper with a reference to the actual libxml2 node. Moreover, as we will see, Gdome2 wrapping nodes are "cached" so that only one wrapper is possibly allocated for a given node of the tree, regardless the actual number of live references from the user application to that node.

This approach has the advantage to exploit most of the libxml2 internal structure for the implementation of DOM, with the minimum duplication of code and a relevant save in terms of memory usage: Gdome2 node wrappers are just used to store fields and to provide functionalities not directly available by means of libxml2 nodes only.

Gdome2 implementation of the Node interface consists of a set of functions providing:

- reading access for read-only node attributes;
- read/write access for modifiable node attributes by means of a couple of get/set functions;
- the semantic associated to the corresponding method in the DOM specification.

---

4. http://xmlsoft.org

In particular, the last possibility is either implemented by simply "forwarding" the method invocation to the corresponding libxml2 function over the libxml2 node, if such function is already provided by libxml2, or implementing it from scratch in the case it is missing.

Tree Structure Differences at Gdome2 Layer

As we have already pointed out in the previous section, the libxml2 tree structure is not always sufficient (nor adequate, in some cases) to fully implement the DOM specification.

Every time the user application asks for the first time an handle to a particular document node, Gdome2 creates a wrapping structure with an internal reference to the corresponding libxml2 node. Elements, attributes and, in general, all the entities derived from Node share the same structure as generic nodes: the inheritance mechanism implemented in Gdome2 allows the sharing of common methods. At this point Gdome2 returns an "opaque" pointer to a GdomeNode structure thus preventing any potentially dangerous modification to the internal fields used by Gdome2.

The internal structure for a node is the following:

```
typedef struct _Gdome_xml_Node Gdome_xml_Node;
struct _Gdome_xml_Node {
    GdomeNode super;
    int refcnt;
    xmlNode *n;
    GdomeAccessType accessType;
    Gdome_xml_ListenerList *ll;
};
```

Gdome2 uses a reference counting mechanism to keep track of the number of users (or live references) of a given node: the structure is shared by all the users which asked for the same DOM node, and the structure is eventually freed as soon as the counter reaches 0.

The sharing of this structure is implemented in an efficient way exploiting the _private field inside libxml2 nodes: when a Gdome2 node is requested by the user, a first check is made on the _private field of the corresponding libxml2 node. If this field is NULL, then the node is accessed for the first time, Gdome2 allocates and initializes a new wrapping structure and sets a pointer to it in the _private field. On the other hand, if _private is a non-NULL pointer, then Gdome2 assumes that it is a previously allocated Gdome_xml_Node and simply returns its value, after having incremented the reference counter accordingly.

However, there are two cases where libxml2 structures are not suitable for this kind of handling:

DocumentType following the DOM specification, this node is meant to provide the lists of entities and notations that are declared both in the external and internal DTD (Document Type Declaration) of the document. However, libxml2 builds different hash tables depending on whether the entities or the notations come from the internal or external fragment of the DTD, for a total of 4 different hash tables. So, when a handle to a DocumentType node is asked for, Gdome2 builds two new hash tables, one for the entities and one for the notations, resulting from the merging of the paired hash tables for entities and notations.

Notation for historical reasons, in libxml2 this node has a particular and too simple structure. So, to treat this node uniformly with all the other node types (in particular for the implementation of the DocumentType interface), Gdome2 allocates a further wrapper to the libxml2 Notation node, so that it looks like all the other nodes.

Node Collections

One of the most important interfaces of the DOM Core Specification is the NodeList interface, which is used to handle ordered lists of Nodes such as the children of a Node, or the elements returned by the getElementsByTagName method of the Element interface. Similarly, there is also a NamedNodeMap interface, used to handle unordered sets of nodes referenced by their name attribute, such as the attributes of an Element. One of their main characteristics is that they are "live" structures, that is, changes to the underlying document structure are automatically reflected in all relevant NodeList and NamedNodeMap objects. For example, if a DOM user gets a NodeList object containing the children of an Element, and he subsequently adds more children to that element, then the added children are silently added to the NodeList, without further action requested from the user side. Of course, changes to a node in the tree are reflected in all references to that Node in NodeList and NamedNodeMap objects.

NodeList Implementation Details

The NodeList interface provides the abstraction of a live ordered collection of nodes.

A first way to implement the NodeList interface is to physically represent it as list of node handles. To make it live we would have to create a list of active NodeLists associated to the Document or to the DOMImplementation objects that own them. Then, any function modifying in some way the tree structure has to update accordingly all the lists where the modified nodes appears. This implementation has the great advantage to be an actual list, so that scanning and searching can be performed quickly (in particular, with regard to the getElementsByTagName method). However, the bookkeeping required to maintain the lists is not feasible, especially if the document is going to be modified often.

The alternative way, which is that implemented in Gdome2, tries to reduce the memory occupation by using a completely "lazy" structure. More specifically, when a NodeList is requested, we return a reference to the following structure:

```
typedef struct _Gdome_xml_NodeList Gdome_xml_NodeList;
struct _Gdome_xml_NodeList {
  GdomeNodeList super;
  int refcnt;
  GdomeNode *root;
  GdomeDOMString *tagName;
  GdomeDOMString *tagURI;
  GdomeAccessType accessType;
};
```

Among the interesting fields of this structure we have a reference to a `GdomeNode` structure called `root` pointing to the root of the subtree of concern (each `NodeList` is relative to a particular subtree, for efficiency reasons. In the worst case, `root` is the topmost document node, corresponding to the whole tree). `tagName` is a reference to a `GdomeDOMString` which is the local or qualified name to be matched; when this field is non-`NULL`, the list includes any element with the given name which is a descendant of the specified root element. On the other hand, when the field is `NULL`, the list only includes the *children* nodes of the `root` element. `tagURI` is a reference to a `GdomeDOMString` representing the namespace URI to be matched. As for the previous field, it can be `NULL` for lists which contains nodes regardless the namespace.

Note in particular that no other structure aside `Gdome_xml_NodeList` is allocated: when the user calls methods of the `NodeList` interface we simply traverse the DOM tree starting from the `root` node, possibly applying the filter represented by the `tagName` and `tagURI`, if specified. In a sense, this implementation of `NodeList` is much like a *live filter* on the DOM tree.

Unfortunately, it is relatively complex to find some effective optimizations for this structure, due to its "liveness" property because, in general, access time is linear in the length of the list. This might induce us to introduce some non-standard interfaces for a more efficient retrieval of document nodes in a near future.

However, note that every node lists whose root node is read-only is defined once and for all at the moment of its creation, so that effective optimizations can be done. In particular, access time can be reduced from linear to constant in the case of an un-filtered node list.

NamedNodeMap Implementation Details

Objects implementing the `NamedNodeMap` interface are used to represent collections of nodes that can be accessed by name, namely attributes inside elements, entities and notations. `NameNodeMaps` objects are not maintained in any particular order, even though they can be accessed by a sequential index for an easy scanning. Like `NodeList` objects, `NamedNodeMaps` are "live".

The `NamedNodeMap` interface is implemented in Gdome2 with the same philosophy of `NodeList`. When the user asks for a `NamedNodeMap`, Gdome2 initializes a structure which contains only the information to locate nodes belonging to the `NamedNodeMap` requested.

The DOM specification requires that nodes contained in a `NamedNodeMap` have to be all of the same type. Moreover they can only be instances of the `Attr`, `Entity` or `Notation` interfaces. The main problem is that libxml2 uses two different structures to maintain these nodes: entities and notations are stored in hash tables, while attributes are stored in double linked lists. Gdome2 correctly handle these differences by setting a flag inside the `GdomeNameNodeMap` structure, so that the correct methods for searching are invoked.

GDOME2 VS. XERCES-C++

In this section we draw a comparison between Gdome2 and Xerces [Xerces] with respect to performances and memory occupation. Xerces is a XML parser available for both Java and C++ programming languages. Here we choose the C++ version because we believe that it is more significant for a comparison with our implementation: C++ is compiled directly into native code, and Xerces-C++, like Gdome2, uses reference counting, while Xerces-Java relies on the garbage collector of the Java virtual machine.

For the benchmarks we used 4 different well-formed XML files generated by a random procedure. In the following table we summarize memory occupation results for Xerces and Gdome2. All the sizes are in Mb (Mega-bytes).

For Gdome2 we provide two values for each test, because the actual memory occupation varies depending on the number of *live references* to DOM nodes. Indeed, as we have explained in a previous section, Gdome2 wrapping structures are allocated only if there is at least one live reference to them. The minimum size corresponds to the size of the libxml2 tree only (no live references), while the maximum size corresponds to the situation where each node in the tree is referenced, so that each node has a wrapping structure allocated.

Thus, the real memory occupation for Gdome2 some-

Memory Occupation

Test	XML Size	Nodes	Xerces Size	Gdome2 Min. Size	Gdome2 Max. Size
A	1	161320	15.6	14.5	19.4
B	2	307921	29.5	27.4	37
C	4	619556	59.2	55.1	74.4
D	8	1175477	112.2	104.5	141.2

how depends on the kind of application using the library. For example, if the application is just making a traversal of the tree looking for a particular node or parsing a configuration file, then the actual occupation will be the minimum size augmented by some amount proportional to the maximum depth of the tree.

The extra space required by Gdome2 is justified by several reasons: first of all, there is a sensible waste of memory due to the memory allocation policy. In particular, for each Gdome2 wrapper we have estimated the waste amount to nearly 8 bytes. This waste can be reduced adopting a more clever allocation strategy, for example allocating several wrappers at a time. Secondly, the Gdome2 architecture leads to

an intrinsic overhead due to the separation of the tree built by libxml2 and the wrapping nodes: libxml2 nodes and Gdome2 wrappers are linked by a couple of pointers, resulting in 8 extra bytes in comparison with Xerces architecture. Finally, we realize that some information encoded in Gdome2 wrappers can be stored in a more efficient way. For example, the `accessType` field is only used to prevent write access to read-only nodes. Thus, this information could be encoded as a single bit in some other field rather than a whole word on its own.

In the following table we summarize parse and visit times for Xerces and Gdome2 for the same XML files. All times are in seconds:

Performances

Test	Xerces Parse	libxml2 Parse	Xerces Visit	Gdome2 Visit
A	3.8	0.9	1.8	0.8
B	7.2	1.6	3.5	1.5
C	14.5	3.3	7.1	3
D	32.3	9.2	117.7	6.8

The table shows that the libxml2 parser is much faster than Xerces'. Moreover, Gdome2 is more than twice as faster as Xerces to complete a full visit of the DOM tree. The visit time for Xerces in test D is not meaningful, because it has been affected by memory swapping due to the tree size which was almost as large as the whole main memory available (128Mb). Incidentally, this also shows that memory occupation can be a critical issue and that the layered architecture of Gdome2 can benefit from it (as in this case) or can be a serious problem.

A possible rationale for the better performances of Gdome2 is that Xerces uses so called *smart pointers* in order to facilitate memory management and to avoid the programmer being aware of reference counting. The point is that smart pointers are a live objects: each time a node is returned as the result of a method in some Xerces class, a fresh smart pointer is allocated (or re-initialized). By contrast, in Gdome2 just a pointer to a node is returned, but the programmer is required to handle reference counting explicitly. Nonetheless, we consider this comparison fair, despite the use of smart pointers in Xerces, because, in Gdome2, visit time also includes memory allocation and initialization of the wrapping structures.

CONCLUSIONS AND FURTHER DEVELOPMENTS

In this paper we presented some aspects of the recent work done in Gdome2, the Gnome DOM Engine. We described some issues regarding the C interface, such as naming and parameters conventions, non-standard features of the library and the object model adopted in Gdome2. We also described some of the major implementation issues of the library, trying to give the reader some flavor of the overall architecture. Fi-

nally, we drew a comparison of performances and memory occupation between Gdome2 and Xerces (for C++) showing that Gdome2 is much faster, thought in some cases it has a larger memory occupation due to its layered architecture.

A lot of work has still to be done. First of all, we plan to implement the Events module, which is going to become a crucial component for editing and authoring purposes. Then, we expect to receive a large amount of feedback from developers who decide to design their applications around XML and the Document Object Model. This will give us a lot of hints for possible optimizations and further developments of the library.

As a concluding remark, it is obvious that the diffusion of XML and its final success strongly depend on the availability of tools for its fruitful exploitation. Gdome2 is aimed to be, after all, a contribution in this direction.

We want to thank Andrea Asperti for many helpful discussions. We also thank the peer reviewers which provided us with some important feedback about the structure of the paper.

BIBLIOGRAPHY

[**HUN00**] D. Hunter et al., "Beginning XML", Wrox Press Inc., June 2000.

[**XML00**] "Extensible Markup Language (XML) 1.0", Second Edition, W3C Recommendation, 6 October 2000, **http://www.w3.org/TR/2000/REC-xml-20001006**

[**UTF-8**] F. Yergeau, "UTF-8, a transformation format of ISO 10646", IETF (Internet Engineering Task Force), RFC 2279, **http://www.ietf.org/rfc/rfc2279.txt**

[**Unicode**] The Unicode Consortium, "The Unicode Standard", version 3.0, Frebruary 2000, **http://www.unicode.org/unicode/standard/versions/Unicode3.0.html**

[**LEVDOM99**] R. Levien, "Gnome World DOMination", 14 April 1999, **http://www.levien.com/gnome/domination.html**

[**DOM**] "Document Object Model (DOM) Level 2 Specification". Version 1.0, W3C Recommendation, 13 November 2000. **http://www.w3.org/TR/2000/CR-DOM-Level-Core/ http://www.w3.org/TR/DOM-Level-2-Events/ http://www.w3.org/TR/DOM-Level-2-Traversal-Range/**

[**C**] B.W. Kernighan, D.M. Ritchie, "The C Programming Language", Second Edition, Prentice Hall, June 1988.

[**Java**] K. Arnold, J. Gosling, D. Holmes, "The Java Programming Language", Third Edition, Addison-Wesley, June 15, 2000.

[**CPP**] B. Stroustrup, "The C++ Programming Language", Special Edition, Addison-Wesley, February 15, 2000.

[**encoding**] D. Veillard, "Libxml Internationalization Support", **http://xmlsoft.org/encoding.html**

[**LEVTXT99**] R. Levien, "Gnome-Text API Documentation", July 10, 1999, **http://www.levien.com/gnome/gnome-text.html**

[**LEVPA99**] R. Levien, "Pango Proposal", rev 0.1, July 28, 1999, **http://www.levien.com/gnome/pango-0.1.html**

[**Names99**] "Namespaces in XML", W3C Recommendation, January 14, 1999, **http://www.w3.org/TR/1999/REC-xml-names-19990114/**

[**Xerces**] "Xerces-C++ Parser", Version 1.4.0, **http://xml.apache.org**

BIOGRAPHY

Paolo Casarini was born in Italy in 1974. He is a graduating student in Computer Science at the University of Bologna. His master thesis is about a DOM Level 2 implementation for Gnome, compliant to the Core and Events modules. He is currently the maintainer of the Gdome2 project. His main interests are focused to the optimization of standard ways for the management of structured data.

Contributions to the Project: He is the supervisor and maintainer of the Gdome2 Gnome module HELM, as well as its main developer.

Luca Padovani was born in Italy in 1974. He won a Ph.D. fellowship in Computer Science in 1999. He worked on design and implementation of distributed concurrent languages with mobile agents. His current interests cover Web Technologies, Web Publishing, concurrency models and functional languages.

Contributions to the Project: He contributed to the design and development of the Gdome2 module with Paolo Casarini. He has designed and implemented the Gtk-widget for the rendering of MathML presentation documents which is going to use Gdome2 as its underlying DOM framework.