Chemical Markup, XML and the World-Wide Web. 2. Information Objects and the CMLDOM

Peter Murray-Rust*,† and Henry S. Rzepa‡

Virtual School of Molecular Sciences, School of Pharmaceutical Sciences, University of Nottingham, London SW7 2AY, U.K., and Department of Chemistry, Imperial College of Science, Technology and Medicine, London, SW7 2AY, U.K.

Received December 8, 2000

We describe the development of a structured method of representing chemistry on the World-Wide Web using an object-oriented approach to information objects. We show how a document object model (DOM) for chemistry can be constructed using as its basis Chemical Markup Language (CML). Application of the CMLDOM to the development of chemical tools is described.

INTRODUCTION

Markup Languages created using eXtensible Markup Language (XML) are becoming accepted as the medium for passing generic and domain-specific information over networks. In many cases this information represents persistent information objects which can be reused in other applications and other contexts. Such messages and objects are likely to become the medium of chemical information flow, replacing the incompatible “legacy file formats” currently used in most applications.

In the first article of this series[1] we described the development of Chemical Markup Language (CML) as a tool for providing the core features required for chemical information. That article described how to create structured documents including chemical and generic information and how to represent them in the concrete XML syntax. Although such documents have value as human-readable and future-proofed files, their main use is likely to be as input and output to computerized processes. It is not always appreciated that a Markup Language in a specialized domain is often of little value without conformant software tools. A very common approach is to specify and implement a Document Object Model (DOM) for a language. Because the XML effort has generated a very large number of DOM-based tools,[2] such representations are extremely powerful and productive, and we review here a number of possible operations.

The creation of a DOM requires a precise and complete understanding of the machine representation of an object (such as “molecule” in chemistry). It therefore requires the use of a defined abstract data structure for the objects, their functionality, and interrelationship. This task is often undertaken by members of the Object Management Group (responsible for the CORBA standard). Although a macromolecular specification has been recently received,[3] there was no preexisting agreement for “small molecules”. We were invited to collaborate on the submission of such a proposal, and the work described here is therefore the first formal publication of an object specification for chemistry. This has implications beyond XML, and CMLDOM can now be used as a primary means of representing chemical information in computers.

Representational Terminology and Language Bindings. We assume some initial familiarity with object oriented programming (OO) and XML and CML terminology (see the Annex for further details). We have benefited greatly from the specification and implementations of the markup languages for mathematics (MathML[9)] and vector graphics (SVG[4]) published by the W3C. This article takes these DOMs as exemplars, and our general approach is specifically chosen to be consistent with them. As in them, the language examples are given in Java except where noted, but can be translated into JavaScript (ECMAScript).

Data Structures and Code Reuse. In his classic description, Knuth[5] proposed that **Programs = data + algorithms**. The essence of this is that good programming technique requires a carefully thought-out data structure to which algorithms can be applied. While we are sure that much chemical software is well designed, it is uncommon for programs to expose a clear data structure and this prevents the reuse of algorithms. Many authors will have to reimplement algorithms simply to fit their data structures, which is costly, unproductive, and error-prone. It detracts from the goal of a component-based software industry.

Modern emphasis is very strongly on reuse, i.e., that a design or implementation once created should be reuseable in another context with little or no rewriting or adapting. Ideally this leads to “information components” and “code components” which have precisely defined interfaces and where the internals of the implementation or design are not exposed or accessible. This is common in numeric and statistical methods where standard libraries (e.g. NAG, IMSL) are used to provide efficiency and accuracy. A programmer has the choice of many algorithms for matrix methods and can choose on the basis of their publicly described functions

* Corresponding author phone: +44 1223 336432; e-mail: pm286@cam.ac.uk.
† University of Nottingham.
‡ Imperial College of Science, Technology and Medicine.

10.1021/ci000404a CCC: $20.00 © 2001 American Chemical Society
Published on Web 07/25/2001

and performance. In chemistry, however, such algorithms have never hitherto been componentized, largely because the data structures for chemical objects are not universally agreed. Indeed the plethora, often inadequate, chemistry "file formats" reflects the lack of a coherent extensible data structure. We believe the time has now come where there is a need for open approaches to chemical information components.

XML makes a considerable contribution to software design because it can describe and document the data structures. Thus an XML document represents a tree in which the component nodes and attributes are precisely defined and to which any tree-based algorithm (enumeration, sorting, walking, merging, pruning, etc.) can be readily defined. Because trees are common and useful, the W3C XML activity[2] has created an Application Programming Interface (API) for XML trees called the Document Object Model (DOM). This encourages standardization of terminology, documentation, and algorithms. We believe this model can form the core for component-based computing and describe here its extension to chemistry.

THE SEMANTIC WEB OF CHEMISTRY

Our dream is for a "Semantic Web" for chemistry. This concept[6] epitomizes the ability for a *machine* to receive a document over a network *without prior agreement on its contents* and to carry out *meaningful processing without human direction*. The sender of the message has to represent the following: (a) what the components of the document are, (b) the document semantics, or how the components are to be processed, and (c) the ontology used, i.e. what the components mean.

XML provides a solution for the first in that the encoding, markup, structure, and namespace is precisely defined. It deliberately does not address the others.

There is no universally agreed mechanism or sets of mechanisms for encoding semantics; the most common Web-based method has been to associate executable code such as plug-ins or Java applets which behave in a predetermined way when installed on the recipients' machines. Agent technology and CORBA are alternative approaches but are normally only deployed for projects where there is much prior agreement between senders and recipients. In this article we propose a Java interface as a first step.

We shall address ontology in a future article and use *molecule*, *atom*, *bond*, and *electron* as concepts in an interface whose ontological description is deferred till then.

Creating, Reading, and Editing XML/CML Documents. Creating systems for processing structured documents is often harder than appreciated. It requires the following:

• a precise, agreed specification that has been tested through prototype implementations;

• tools to write compliant documents;

• tools to read compliant documents;

• tools to modify (edit) and validate compliant documents.

If the specification is flexible, a read/write/edit tool must cater for all possible variants of documents. This requirement is a major challenge. We argue here that the solution is to implement such a tool with a DOM and that a CMLDOM is critical to *chemical interoperability*.

A Formal Approach to Describing Objects. The Unified Modeling Language (UML)[7] is rapidly becoming the de facto method for representing OO concepts. Most implementations have graphical interfaces and provide tools for supporting the creation of software. For example, a framework described in UML can be converted to an IDL (Interface definition language) or XML representation or to Java class libraries.[8] Although XML is not directly an Object-Oriented approach, it is easy and common to map it onto OO frameworks. XML can support most OO concepts, the commonest being the following:

• **Abstraction.** The formalization of the essential properties of a (real-world) entity: The abstraction is usually a subset of most human perceptions of the real-world. For concrete subjects such as chemistry, XML elements normally map well onto these formalizations. Thus elements in CML represent a core set of universal self-consistent properties of molecules, atoms, etc. from which we create corresponding OO abstractions.

• **Encapsulation.** Each object is a "black box" and clients of it only have access through the interface. The implementation is not specified, and details cannot normally be obtained from the interface which exposes components through methods. The interface is often tied closely to the XML elements and their attributes, though some of the methods are related to XML prose descriptions.

• **Aggregation.** An Object contains other objects (a "hasA" relationship). This maps directly onto the content model of XML DTDs (Document type definition), so that a DTD defines the object aggregation model. For CML, a <molecule> element can directly or indirectly contain elements such as ⟨*atom*⟩, ⟨*bond*⟩ etc. and a Molecule object can contain Atom objects, Bond objects, etc.

• **Polymorphism.** Objects that share a common interface can be used in identical ways by other objects. Thus all markup languages share common features such as XMLAttributes through XML itself. For example, MathMLDOM Objects (the W3C's specification for mathematics)[9] and CMLDOM objects both implement the W3CDOM interface,[10] so either can be "plugged into" a generic W3CDOM-compliant tool.

• **Inheritance.** An Object can inherit properties from another (its superclass) and can modify or add to them. Inheritance can be from a single superclass or many ("multiple inheritance"); Java has eschewed multiple inheritance and so do we, using interfaces where appropriate. XML has no direct concept of inheritance, though the recent XMLSchema specification[11] provides some of this functionality, with multiaxiality through facets. CMLDOM is designed for inheritance as outlined below.

Other properties, which XML does not normally support are as follows:

• **State** and **Sequence.** These relate to dynamic processes. They are not currently used in CML.

• **Behavior.** This is normally provided through methods. XML is, at heart, a language for describing static documents, though many specific markup languages have found the need to specify behavior. This is true for chemistry, but we defer the detailed behavior-related methods to a future ontological article.

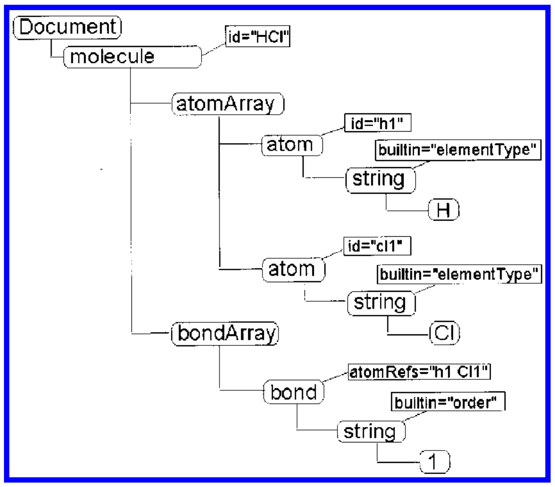

Figure 1. An in memory representation of a Document Object Model. Elements are shown in rounded boxes, Text in leaf rounded boxes and attributes in rectangular boxes. The Document contains a single rootElement (elementName "molecule") with a single attribute (name "id" value "HCl"). This element has two children (atomArray and bondArray) which are siblings. Each contains further descendants (atom, bond, string, etc.) all with attributes. The string elements each contain a TextNode; these have characterData values (e.g. the first has value "H"). The tree can be traversed in document order, starting with Document and continuing molecule, atomArray, atom, string, "H", atom, string, "Cl", bondArray, bond, string, "1".

Since CML V1.0 does not formalize the behavior of CML objects, the CMLDOM interface will be the major way of achieving this.

THE CMLDOM

The DOM acts as a central abstract representation of any document or object, *with complete removal of syntactic information*. An XML document in canonical form[12] can be transformed into a corresponding DOM without loss of information, and the DOM can be retransformed (*serialized*) back to the document in XML syntax.

XML supports recursive tree-structured containment by and of elements. A DOM is an "in-memory" representation of the complete document and allows the user access to any part of the information at any subsequent time. The DOM does not actually mandate the abstract data structure, but we shall assume a tree-structure. A DOM is described by a *Document Node* that contains further *Nodes*, in a tree structure isomorphous to the XML document. There are 12 types of Node, of which we only consider *Document*, *Elements*, *Attributes*, and *Text* in this article. Since an XML document may only have one *root Element*, so an XML DOM only has one root *Element* Node, which is the *child* of the Document Node. An *Element Node* may have *Attribute Nodes* each with a *name*, *value*, and (optionally) a *type*. Figure 1 shows a simple XML document (the formula and connection table for molecular hydrogen chloride) and its abstract representation in a DOM.

This diagram shows that

• A node may have 0 or 1 *parent* Nodes.

• A node may have 0 or many *siblings* either or both *elder* and *younger* (i.e. order is important).

• A node may have 0 or many children.

• A node may be a Document, an Element, or a Text.

• An Element may have 0 or many Attributes, which are not child Nodes.

• An Element must have a *tagName* (*elementTypeName* in XML).

• A Document may have 0 or 1 children. A well-formed XML document has exactly 1 element child of Document.

• A Text has *character data content*, which may be the empty string "".

• An Attribute must have a *name* and a *value*, which may be the empty string ("").

• An Attribute may have a *type*. For attributes with type *ID* all values must be unique within the document.

The DOM represents the XML document **after** processing by the parser, i.e., all defaults and type information contained in a DTD or Schema will now be explicitly present. The tree holds the information in *document order*. Thus a *traversal* of the tree in *pre-order mode* will visit nodes in the correct order to reproduce the document. The order of attributes within an Element is arbitrary.

The primary benefits of using the DOM model are that the whole tree is available for analysis at any time, and it can be processed by "walking" (traversal). A simple way to do this is to extract all the childNodes of an element, process them, and process their childNodes recursively. Typical Java code (adhering to the W3C DOM)[10] for checking bonds in (nested) MoleculeNodes is shown below.

```
public class MoleculeNode extends org.w3c.dom.Node {
public void processNode() {
        this.checkBonds();
        NodeList childNodes = this.getChildNodes();
        for (int i = 0; i < childNodes.getLength(); i++) {
                Node childNode = childNodes.item(i);
                if (childNode instanceof MoleculeNode) {
((MoleculeNode)childNode).processNode();
                }
        }
}
private void checkBonds() {//special processing for MoleculeNode}
}
// end of class MoleculeNode
// in user program
        Document document = new DocumentImpl("myProtein.xml");
MoleculeNode molNode = document.getMoleculeNode(count);
molNode.processNode();
```

This will recursively process molNode. The childNodes are extracted and traversed. Only child Nodes of class MoleculeNode are further processed, with other Nodes and Text ignored. Programs using the DOM approach will frequently have constructs of this sort.

DOMs can be constructed by two main methods. An existing XML document is read in and converted to a DOM, using one of the many generic tools available for this.[13] Given the CML document

```
<molecule id="HCl">
<atomArray>
    <atom id="h1">
      <string builtin="elementType">H</string>
    </atom>
    <atom id="cl1">
      <string builtin="elementType">Cl</string>
    </atom>
  </atomArray>
  <bondArray>
    <bond atomRefs="h1 cl1">
      <string builtin="order">1</string>
    </bond>
  </bondArray>
</molecule>
```

a DOM-aware parser would create the in-memory representation shown in Figure 1. Alternatively, an empty Document is constructed, and Nodes are added to it in document order with the function appendChild(Node). To create the example

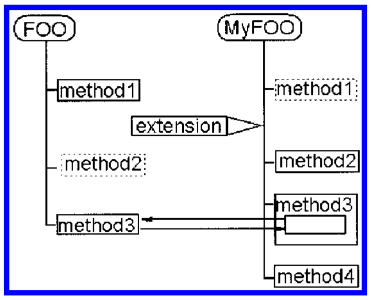

Figure 2. Extension of FOO to MyFOO. method1 is explicitly provided in FOO but not in MyFOO. MyFOO inherits the functionality from method1 in FOO. method2 is explicitly provided in both FOO and MyFOO. method2 in MyFOO overrides the functionality from method2 in FOO (i.e. completely replaces it). method3 is explicitly provided in both FOO and MyFOO. As above the latter overrides the functionality but does not destroy it. It can therefore make a call to the superclass method to extend or modify the functionality of method3 FOO rather than replacing it. method4 is not provided in FOO and is unique to MyFOO, so it must appear in its interface.

document in Figure 1 we could write

```
// make a new Document
Document document = new DocumentImpl();
// make an element for the molecule and add to the document
Element moleculeNode = new ElementImpl("molecule", document);
moleculeNode.setAttribute("id", "HCl");
document.appendChild(moleculeNode);
// add atomArray element
Element atomArray = new ElementImpl("atomArray", document);
molecule.appendChild(atomArray);
// add atoms to this
Element hydrogen = new ElementImpl("atom", document);
hydrogen.setAttribute("id", "h1");
atomArray.appendChild(hydrogen);
Element elementType = new ElementImpl("string", document);
elementType.setAttribute("builtin", "elementType");
hydrogen.appendChild(elementType);
Text hText = new TextImpl("H", document);
elementType.appendChild(hText);
Element chlorine = new ElementImpl("atom", document);
hydrogen.setAttribute("id", "cl1");
atomArray.appendChild(chlorine);
elementType = new ElementImpl("string", document);
elementType.setAttribute("builtin", "elementType");
chlorine.appendChild(elementType);
Text clText = new TextImpl("Cl", document);
elementType.appendChild(clText);
// add bondArray
Element bondArray = new ElementImpl("atomArray", document);
molecule.appendChild(bondArray);
Element bond = new ElementImpl("bond", document);
bond.setAttribute("atomRefs", "h1 cl1");
bondArray.appendChild(bond);
// create and add bond order
Element order = new ElementImpl("string", document);
order.setAttribute("builtin", "order");
bond.appendChild(order);
Text oText = new TextImpl("1", document);
order.appendChild(oText);
```

This may appear long-winded, and convenience methods are often used to package repetitive constructs. Note that Elements and Text must be created in a document context, i.e. they always have an Owner Document. The interfaces (e.g. Element and Text) must be *implemented* by classes such as ElementImpl and TextImpl. This might be expressed as

```
public class ElementImpl implements org.w3c.dom.Element {
// constructor with tagName and Document
        public ElementImpl(String tagName, org.w3c.dom.Document);
}
```

CMLDOM and Extending the DOM Functionality. The examples above use the classes defined in the DOM interface and although they describe CML documents, their classes have no chemical functionality. We have developed CMLDOM as an *extension* of the W3CDOM. Extension, a process also called *subclassing* or *derivation*, as a standard approach for creating specialized classes from more general ones. A CMLDOM class *inherits* all the methods of its W3CDOM *superclass* or base class. The W3CDOM describes 17 interfaces, but only two need *specialization* by extension in CMLDOM:

```
org.xmlcml.cml.CMLDocument extends org.w3c.dom.Document
org.xmlcml.cml.CMLBase extends org.w3c.dom.Element
```

The Java class names can be determined uniquely across the planet by the use of reversed domain names and extensions for the *packages*. For simplicity we shall omit the package-name from this point.

CMLBase then acts as the superclass for many of the CML interfaces. All the remaining W3CDOM classes are potentially usable by CML interfaces, the most important being

```
org.w3c.dom.Attr;
org.w3c.dom.CDATASection;
org.w3c.dom.Comment;
org.w3c.dom.Document;
org.w3c.dom.Element;
org.w3c.dom.EntityReference;
org.w3c.dom.NamedNodeMap;
org.w3c.dom.Node;
org.w3c.dom.NodeList;
org.w3c.dom.ProcessingInstruction;
org.w3c.dom.Text;
```

The power of inheritance is shown in Figure 2.

Through this approach CMLDOM inherits all the basic functionality of W3CDOM which provides access (*getters* and *setters*; see below), traversal, editing, and much more. CMLDOM is therefore completely consistent with any W3CDOM-compliant software and can benefit from generic tools. Its power comes mainly from extension through new methods rather than modifying W3CDOM functionality.

CMLDOM is *extensible*. We have constructed a core CMLDOM (V1.0) which represents the chemical objects (XML elements) specified in the CML DTD V1.0. This core primarily represents the data structure of CML objects, allowing access for reading and writing (*get* and *set*, v.i). However a DTD does not specify other *methods*, and so a DOM will often contain "non-DTD" methods to provide the domain-specific functionality. This is best provided through extension (subclassing), and so we are currently extending the core DOM interface to create additional packages which supply non-DTD methods. The distinction is shown through examples (see also "non-DTD" section below):

```
org.xmlcml.normalize.NormalAtom extends org.xmlcml.cml.Atom
org.xmlcml.style.StylableMolecule extends org.xmlcml.cml.Molecule
```

Design Features of the Core CMLDOM. Accessors and Mutators ("Getters and Setters"). Given a (simplified) part of the CML DTD such as

```
<!ELEMENT atom (string | float | integer)*>
<!ATTLIST atom id CDATA #REQUIRED>
```

the following methods provide ways of accessing the content and attributes:

```
public class Atom extends CMLBase {
        public String getString();
        public void setString(String s);
        public double getFloat();
        public void setFloat(double d);
        public int getInteger();
        public void setInteger(int i);
        public String getId();
        public void setId(String id);
```

These types of construct account for much of the CMLDOM interface. All CML elements have attributes and most have child elements in their content model. There will normally be a get method for each, and often a set method. set methods will not be present where the information is not normally alterable by the programmer, particularly if internal consistency can be lost.

Interfaces for Common Attributes. Since many classes represent elements with an id attribute, it is useful to define an interface that they all implement

```
public interface hasId {
        public String getId();
        public void setId(String id);
}
```

and enforce this through the class declaration such as

```
public class Atom extends CMLBase implements hasId {...}
```

Many classes implement several interfaces corresponding to widely used attributes in the DTD.

```
HasConvention;
HasCount;
HasDictRef;
HasId;
HasTitle;
HasUnits;
```

Accessing Child Elements. Some of the containment relationships in CML are more complex than the DTD can imply. The array elements (stringArray, floatArray, integerArray) require access to their components, enforced by

```
public interface HasSize {
        public int getSize();
        public StringVal elementAt(int i);
        public Vector getStringVector();
        public void setDelimiter(String delimiter);
        public String getDelimiter();
}
```

Angle, Bond, and Torsion objects have to refer to atoms for their definition, enforced by

```
public interface HasAtomRefs {
        public Atom getAtomRef(int serial);
        public Vector getAtomRefVector();
        public void addAtomRef(Atom atom);
        public void deleteAtomRef(Atom atom);
}
```

Multiple Children. get and set are only appropriate for attributes or where a child can occur 0/1 times. Where many children of the same type can occur, access is usually through add and delete rather than set. The whole set of children is available through the DOM's getChildNodes() method, but this does not distinguish Elements from Text and the programmer has to iterate and filter the NodeList. We therefore provide a number of convenience methods to access children of a given type, usually returning a Vector. Note that Vector does not enforce classes for the contained elements, and these must be explicitly cast on extraction.

Although XML preserves all order in an element's children, CML insists on this only where it makes chemical sense. For example

```
<atom id="a1">
  <float builtin="occupancy">0.5</float>
  <string builtin="elementType">O</string>
</atom>
```

and

```
<atom id="a1">
  <string builtin="elementType">O</string>
  <float builtin="occupancy">0.5</float>
</atom>
```

are semantically equivalent. We would discourage implicit semantics of order. Thus start and end of bonds, or product and reactant molecules, should be represented by markup and not by order in the document.

Atoms in particular should never be identified by their position but always by an unchanging unique ID. CML does not prescribe the form of this ID or the method of uniquification. Some tools may use hashcodes based on time, userID, etc. while others may have implicit semantics (e.g. cl2, ribose-c3', etc.). There is a difficulty when independent documents are merged and the IDs may collide. One approach would be to base IDs on URL-like syntax to ensure the planet-wide uniqueness of such IDs. Others may wish to convert IDs as part of the merging process. We have not prejudged the solutions that may emerge from the W3C and would only emphasize here that IDs must be unique within molecule context and may benefit from being unique within a wider one (document- or enterprise-wide).

Where multiple children of the same type can occur the accessor methods must allow for modification (addition, removal) from an ordered Array (Vector) of such children.

builtin Attributes. builtin attributes provide great flexibility but require considerable structure in both classes and interfaces. Atom, Bond, and Crystal objects can all contain child objects with builtin attributes and all are subclassed from AbstractBuiltinContainer, which provides support for these. This latter class includes the signatures

```
public interface HasBuiltinNames     {
        public int getBuiltinCount();
        public StringVal getBuiltinVal(int builtinId);
        public void setBuiltinVal(int builtinId, StringVal value);
        public StringVal getNonBuiltinVal(String name);
        public void setNonBuiltinVal(String name, StringVal value);
        public void deleteNonBuiltinVal(String name);
}
```

which itself has the class signature

```
public interface AbstractBuiltinContainer extends CMLBase, HasBuiltinNames {

// normally subclassed to avoid problems with static
       public abstract int getBuiltinCount();
/** extracts all children with builtin attribute; */
       public Vector getBuiltinChildElements();
       public StringVal getBuiltinVal(int builtinId);
       public void setBuiltinVal(int builtinId, StringVal value);
/** gets the SINGLE child with the given builtin */
       public Element getBuiltinChild(int builtinType, String[] builtinNames, int type)
throws CMLException;
/** returns the value of a SINGLE child with given builtin */
       public String getBuiltinChildStringValue(int builtinType, String[] builtinNames)
              throws CMLException;
       public int getBuiltinChildIntegerValue(int builtinType, String[] builtinNames)
              throws CMLException;
       public double getBuiltinChildFloatValue(int builtinType, String[] builtinNames)
              throws CMLException;
       public void setBuiltinIntegerChild(int builtinType, String[] , builtinNames, int
value) throws CMLException;
       public void addBuiltinChildWithFloatValue(int builtinType, String[] builtinNames,
              double value, boolean unique, boolean ignoreExisting) throws CMLException;
```

Many builtins also have convenience methods; thus Atom implements getOccupancy() to save the programmer having to constantly interrogate the builtin children. Molecule, Bond, and Crystal have analogous methods. We have also provided classes for some of the builtin datatypes such as occupancy, isotope, atomParity, etc. as these cannot always be represented by single strings or numbers.

Non-DTD Extensions. We are developing additional packages to provide extensions of CMLDOM1.0. These are not finalized but currently include the following:

• org.xmlcml.query for substructure search. Example: org.xmlcml.query.AtomMatch.matchAtoms(Atom targetAtom, Atom queryAtom);

• org.xmlcml.style for rendering of molecular display. Example:

org.xmlcml.style.AtomStyle.setColor(Color color);

• org.xmlcml.normalize for normalizing the representation of molecules (e.g. from different starting conventions. Example:

org.xmlcml.normalize.NormalAtom.getDoubleBond-Equivalents();

• org.xmlcml.topology for representation of molecular substructures (e.g. ring nuclei and chains). Example:

org.xmlcml.topology.getRingOverlaps();

The description and behavior associated with these is clearly coupled to systematic chemical ontology, and we shall defer these for discussion in a future article.

Other Considerations in Writing XML and CML-Aware Code. CML has been designed as a DTD fragment, i.e., the components can be reused within other document types and namespaces and can also, if appropriate, contain other information. We have already described[14] the use of XML stylesheet transformations (XSLT) and shown how to extract and process the molecular information. XSL can be used to process DOMs, i.e., a stylesheet can convert a source DOM to a result DOM in memory.

For specific chemical operations calls to the DOM are normally required. An example is to extract the molecular weight of a molecule. In some cases, e.g. in SwissProt files, this is included in the raw data file, while in other cases it can be computed from the children of the molecule element. Since molecular weight is a commonly requested quantity it makes sense to provide a method defining this functionality. This can then be implemented in different ways according to the information available, the algorithms requested and the structure of the document.

CML-aware software therefore uses classes corresponding directly to the elements in the DTD, specifically:

```
public class Molecule extends Element implements Molecule
public class Atom extends Element implements Atom
public class Bond extends Element implements Bond
public class AtomArray extends Element implements AtomArray
public class BondArray extends Element implements BondArray
public class Cryst extends Element implements Cryst
public class Sequence extends Element implements Sequence
```

Because these classes implement org.w3c.dom.Element and org.w3c.dom.Node they automatically expose all their interfaces, e.g.

```
public Vector Element.getElementsWithTagName(String name);
```

which returns all descendants of the current Element with a given name. They inherit, but need not use, the data structure used to implement Element. For example Molecule might hold Atoms internally in a Vector rather than as child Nodes. A protein could hold child atoms and bonds in a conventional set of arrays and use specific code to implement getChild-Nodes().

The interfaces such as Molecule provide additional functionality. Examples of this could be

```
public interface Molecule {
// extract the i'th atom regardless of intervening non-Atom nodes
public AtomNode getAtomAt(int i);
// extract an AtomArray (creating from the AtomNodes if required)
public AtomArray getAtomArray()
// extract 3-D functionality if it exists
public AtomArray get3DAtomArray();
// extract 2-D functionality if it exists; else convert from 3D
// functionality if it exists and flag is set
public AtomArray get2DAtomArray(boolean 3Dto2D);
```

The last illustrates the power of interfaces. If builtins for x2 and y2 exist, extract them. If not, and x3 and y3 are present, extract them (i.e. as a 2D projection) if 3Dto2D is set.

Moving down the hierarchy, information can be extracted from AtomArray such as

```
public interface AtomArray {
// extract the i'th atom
public AtomNode getAtomAt(int i);
// extract a vector of Atoms
public Vector getAtoms()
```

and all classes will have such accessors. There will be similar tools for adding or changing contained objects such as

```
public interface Molecule {
// add an atom
public Atom append(Atom atom);
// add the crystallographic information (replacing existing one)
public void set(Cryst cryst);
```

CML implicitly defines primitives such as Point3 (3-D coordinate), and implementations might create ancillary classes to support these such as

```
public class Point3 {
        double xyz = new double[3];
}

public interface Atom {
        public Point3 getPoint3();
}
```

It is useful to check that CML elements contain not only well-formed XML but that the content is chemically meaningful. XML cannot provide this, and so CML interfaces should encourage implementers to check. Some validity checks are syntactic such as

• do the atomRef values in bonds point to valid atoms?

• are atomId values unique?

Others may relate to content, though implementers must be careful not to make “reasonable” assumptions. CML deliberately does not constrain value, and so (for example) negative values of occupancies, unusual valencies, missing elementTypes, etc., although possible mistakes are not required to be flagged as errors.

Interfaces in CMLDOM V1.0. The Annex (section 5) shows the Javadoc output[15] for the current implementation of CMLDOM1.0. The Interface Hierarchy may show a given interface several times but that the Class hierarchy is effectively tree-structured. All classes are in package org. xmlcml.cml unless specified.

Datatyping and Constraints. XML V1.0 provides no support for datatypes other than character data (CDATA and #PCDATA). CML V1.0 was written in this environment, and so the types and constraints on CML components are contained in prose rather than formal language. In CMLDOM we have normally implemented them by one of the three primitive types java.lang.String, int, and double. In Javascript this is effectively reduced to a string and number type. Moreover we have not imposed many additional constraints on values. This is adequate for many simple chemical systems, but we anticipate the need for additional datatypes with additional facets.

The W3C has recently published[11] a *Candidate Recommendation* for XML Schemas. These provide increased power and flexibility over DTDs, such as inheritance, constraints, and datatyping. Their functionality is a “superset” in that DTDs can be algorithmically converted to corresponding Schemas but not *vice versa*. We are actively translating CML functionality and concepts to the Schema-based representation but will preserve the CML V1.0 DTD as a valuable method of representing chemical concepts. This is a complex and fast-moving area, and we shall try to mix usability and simplicity with power. As always we shall not release specifications for which we do not have implementations and proof-of-concept.

Status and Type of builtins. In CML V1.0 we discouraged "tag soup" by deprecating element syntax for the builtin concepts (e.g. not providing <occupancy>, <elementType>). This would have led to a very complex DTD which would break whenever a new concept was added. We also had to choose between the role of the element and its datatype (e.g. <float>). It is clear, however that

```
<atom id="a1"><string builtin="elementType">O</builtin></atom>
```

is semantically (though not syntactically) equivalent to either

```
<atom id="a1"><elementType>O</elementType></atom>
```

or

```
<atom id="a1" elementType="O" />
```

CMLDOM effectively supports any of these, and indeed Atom.getElementType() is most reminiscent of the last two.

The new XML technology (Schemas and XSLT stylesheets) makes it trivial to convert between these representations and to validate such CML documents. Schemas provide for dataTyping of attributes so that all examples can now be constrained to have appropriate values for elementType which was not possible for DTD-based representations. We do not yet envisage variation in serialization of CML, and only the first example is currently valid. However the programmer may find the variety of access methods are valuable.

We also expect that scalar numeric datatypes will be inadequate for many of the builtin concepts. It is valuable to have nullable values (a Schema concept) which int and float do not support. We also expect that symbolic representations (in nonstandard CML) may be useful such as

```
<atom id="a1"><atomParity>+x</atomParity></atom>
<atom id="a2"><atomParity>-x</atomParity></atom>
<atom id="a3"><nonHydrogenCount>1,2</nonHydrogenCount></atom>
```

The first two could respectively signify that atoms a1 and a2 had opposite (but unknown) atomParitys which would denote known relative (but not absolute) stereochemistry. The third could denote an allowed range of hydrogen counts on a3, opening the way to formalization of search queries. In both cases we would require nonprimitive objects such as org.xmlcml.cml.Occupancy and org.xmlcml.AtomParity.

Serialized Output of CML. Serialized XML output (i.e. in XML syntax) is normally carried out by recursive traversal of the DOM in depth-first fashion ("document order"). This is not defined in the W3C DOM, but tools such as the Xerces package from Apache (org.apache.xml and dom.DOMWriter)[16] are excellent exemplars. The CMLDOM implementation requires that the elementTypeName for each class ("tagName") corresponds to that in CML V1.0, i.e. without namespaces. Under these conditions any DOMWriter should create a well-formed CML document. The CMLDOM can be created to enforce validity among the CML components and does this through the access methods, e.g. an Atom cannot have two occupancies if setOccupancy() is used. The CMLDOM V1.0 DTD is much weaker than normal chemical constraints, and we are developing CMLDOM to represent the chemical consistency that users will wish. This is not trivial, since opinions of chemical validity differ and the current CMLDOM is forgiving. Our preferred approach is to constrain the DOM through XML Schemas where possible and enhance them with our own XML-based constraint mechanism. It will be beneficial if some CML objects implement an interface such as

```
public interface Outputtable {
      public void outputXML(java.io.Writer) throws java.io.IOException;
}
```

where Molecule, Atom, etc. should implement Outputtable.

Additional Functionality. We have developed additional software for use with CMLDOM1.0. At present we do not feel this is core to the DOM functionality but may add some of this in later versions. Among the functions are the following:

- connection tables and ring perception
- management of crystallographic coordinates and symmetry
- chemical perception (e.g. aromaticity)
- valence calculations and checking
- calculation of chemical properties

Special mention should be made of substructure searching, which would be expected to be a natural chemical extension to generic XML query functionality. Although many chemical information systems implement such searches, there have been few interchangeable tools supporting it, which we believe is because of the difficulty of representing search and target molecules in abstract form. CML can be extended to support search fragments (e.g. the hydrogenCount and formalCharge values can be represented as ranges) and atomRefs could be lists to support Markush structures. This allows a substructure search to be defined through a function in Molecule such as

```
public Vector matchAgainst(Molecule search);
```

where the Vector is a list of the matched atoms in the "**this**" Molecule. Because of the flexibility of CML it is easy to add functionality to the search molecule, including geometrical constraints and fuzzy quantities. Since all quantities are expressed as CML, searches can be serialized so that a robust interchangeable search description is finally possible.

Legacy Files. An important area is conversion of CML to and from legacy files. In general a legacy file is less structured than CML and less extensible, so that information is lost on transformation from CML to legacy. CMLDOM1.0 has been designed to hold as many commonly used legacy concepts as feasible, and it is always possible to extend CML, for example by using nonbuiltin children with appropriate convention attributes. Thus the MDLMolfile version information can be held as

```
<string title="version" convention="MDLMolfile">V2000</string>
```

without loss of information. CML-aware tools can be asked to search for various conventions and output the information if they hold it. We may provide limited ontological mapping of the information in future versions. At present CML has a simple core ontology for key concepts such as bond order and carries out bidirectional mapping for legacy conversions if possible. Each legacy conversion involves a subclassed CMLDocument

```
public class NonCMLDocumentImpl extends CMLDocumentImpl implements NonCMLDocument
```

and, for example

```
public class MDLMolImpl extends CMLDocumentImpl implements MDLMol
```

Interoperability with Other Markup Languages and DOMs. CML is designed to reuse concepts and objects from other languages. We have made extensive use of the W3C's SVG (for graphical representation) and shown that the languages interoperate without technical problems (in the prototype Java CMLBrowser Jumbo3-J[17]). In some cases we build a hybrid DOM with elements from both languages and traverse with appropriate strategies (e.g. selecting CML or SVG elements according to need). CML defines a modest range of scientific datatypes but is also able to interoperate with markup languages from other disciplines. As yet, none of these have been formally published, but several have active groups. For example CML has been adopted to support chemical components as part of the Materials Markup Language (MatML at NIST). We are also working with the authors of SELFML, a IUPAC/CODATA project to represent physicochemical data; CML will support the formulas and connection tables of the substances.

Other Language Bindings. Although Java is the language of choice for initial XML DOM implementations, conversion to other languages such as C++ is also valuable. The interface can be created directly from the IDL specification.

We have also partially implemented CMLDOM in JavaScript (ECMAScript), the native language of browsers. This allows a CML file to be read directly into a browser's XMLDOM, transformed with Javascript, and rendered using browser interfaces such as HTML and SVG plugins. In this way we have created JUMBO3, a lightweight CML viewer running on any XML-aware browser, which converts CML to SVG primitives to display high-quality molecular rendering on screen, particularly of 2-D structural formulas.[17]

XML can also be formatted for high-quality typesetting through the XSL-FO specification. We have shown[14] how this can be rendered with FOP[18] for print-ready formats such as Acrobat PDF, but the design of FOP also allows for DOMs to be plugged, in so that a high-quality chemical print or screen-based publishing tool can be created.

Event Stream Model (SAX). The DOM is one of two main interfaces for processing XML, the other being the event stream model (SAX API). SAX is a low-level interface and does not define objects. It is often used to support the creation of DOMs, and CMLDOM1.0 has a SAXDOMBuilder class for this purpose. The document is regarded as a series of components (here tags and text) which fire events as the document is traversed. Running text (e.g. in an HTML P(aragraph)) illustrates the concept. A start tag (such as <b>) fires an event which would switch on Bold formatting which remains in force until </b> fires an event. In Example 1 the content of <p> can be as running text as well as a tree. Traversal fires events in the order and with the arguments shown. Note that every element fires both a startElement and endElement (even the *empty* <molecule/> element).

Example 1

```
<p>The <b>very</b> toxic <molecule id="a3412" title="carbon monoxide"/></p>

events:
startDocument();
startElement("p", null);
characters("The ");
startElement("b", null);
characters("very");
endElement("b");
characters(" toxic ");
startElement("molecule", [id="a3412", title="carbon monoxide"]);
endElement("molecule");
endElement("p");
endDocument();
```

The event stream model is particularly useful for large documents. Little memory is required, and the implementer need only select those events which are required for a given purpose. A typical example would be to find all <molecule>s in a set of large documents (e.g. patents, reports, or primary publications).

An implementation of the SAX interface operates through call-backs (events) of which the most important functions are

```
startDocument();
endDocument();
startElement(String name, AttributeList list);
endElement(String name);
characters(char[] ch, int start. Int length);
```

A program to search for <molecule>s and record their IDs might contain

```
public class MoleculeSAX implements org.xml.sax.HandlerBase {
        java.lang.Vector idVector;
        int count;
// various no-op call backs omitted
        public void startDocument() {count = 0;}
        public void startElement(String name, AttributeList list) {
                if (name.equals("molecule")) {
                        String id = list.getAttributeValue("id");
                        idVector.addElement(id);
                        count++;
                }
        }
        public void endDocument() {System.out.println("Count: "+count);}
}
```

This might be the only active code (all other events are no-ops). The full callback structure is not given since there are excellent examples in any SAX distribution.

The attractions of SAX are that it can deal with large problems more efficiently than the DOM. Thus to read a protein structure of 5000 atoms into a DOM where every atom had 20 child Nodes could require 100K nodes and 20+Mbytes of memory. Using SAX the <atom> events can be trapped and the data read into a table or vector. Similarly the <sequence> information (SEQRES in PDB) could be extracted without needing to process the <atom>s. The disadvantages are that SAX events are transient, the whole document is no longer available, and the programmer has to supply context information.

CONCLUSION

CMLDOM represents the chemical version of the emerging spectrum of DOM-based editors. We see it as an integral component of future chemical editors in that the creator of a chemical editor should reuse CMLDOM technology as the core engine. We also see multicomponent publishing tools in STM disciplines which will integrate a variety of DOMs such as MathMLDOM, SVGDOM, and CMLDOM. In this way a set of truly interactive and reuseable documents can be created by the whole chemical community without having to access the XML technology directly.

ANNEX

1. Some Terminology and Concepts. The W3C community describes its interfaces in the modeling language *Interface Definition Language* (*IDL*) from the *Object Management Group* (*OMG*) and also in their concrete representation in Java or ECMAScript (Javascript). Since almost all XML/CML tools are initially written in Java, we have used it as the main binding in this article. It is straightforward to create IDL specification from the Java described here and also to recode into other OO languages (C++, etc.). Although the examples given are primarily taken from CML, the principles are easily extended to other markup languages.

Interfaces are an important feature of languages such as IDL and Java; they allow specification of how to access objects and functionality but leave the details of that implementation to the creator of the software. An API (Application Programming Interface) such as the DOM specifies the classes and methods that an implementer must provide. Java interfaces only document public and protected methods and do not describe what constructors should be provided. A good interface can be thought of as a comprehensively documented "black box" where every possible behavior is anticipated. Thus a **molecule** class might have an interface **getMolecularWeight**(), and the calling program would not need to know whether this was computed initially and cached or recalculated for every call. However the method would have to define the basis of molecular weight calculation (e.g. reference to Oxygen or Carbon). The *accessor* method (e.g. **atom**.**getCartesianX3**()) for getting Cartesian coordinates might first have to check whether fractional coordinates and cell parameters were present and if so create and apply an orthogonalization transformation.

The term *document* is used to describe any well-formed XML instance, and readers should be aware that it can be composed of or contain "data" or "molecules". Where possible we use the term *information object* to describe the abstract structured data. The output of the object as an XML document is a *serialization*.

Note that a document may consist of many distributed components (often XML entities) and will frequently reuse communal information (e.g. a set of molecules in CML). In some cases the entities may have *persistence* (e.g. they can be downloaded and stored), while in others they may be produced as a *transient* stream. A document can be built from smaller documents; thus a complete year of this journal can be regarded as a single document.

The terms *entity*, *attribute*, and *element* are used in their XML sense. ElementType is unfortunately in common use in XML and also as a CML builtin, so it must be determined from context.

2. The XML/CML Information Set. XML documents contain precise information which can be created or extracted, and the W3C has produced a Working Draft of the XML information set.[19] Quoting from that draft: *The XML information set can contain eleven different types of information items (in the following list, read "required" as "required if present in the original XML document"; see also Processor Limitations, below):*

1. a document information item (required)
2. element information items (required)
3. attribute information items (required)
4. processing instruction information items (required)
5. character information items (required)
6. reference to unknown entity information items (required)
7. comment information items (optional)
8. a document type declaration information item (optional)
9. entity information items (required for unparsed entities, optional for others)
10. notation information items (required)
11. attribute declaration information items (optional)

In principle a complex chemical document (e.g. a patent or primary publication) could contain examples of all of these. The current article has been limited to a small subset found in all chemical information objects, but anyone supporting a full chemical editing system will probably need to include facilities to manage most of the other components. The draft emphasizes the difference between an interface and a data model:

"The XML information set does not require or favor a specific interface or class of interfaces. This specification presents the information set as a tree for the sake of clarity and simplicity, but there is no requirement that the XML information set be made available through a tree structure; other types of interfaces, including (but not limited to) event-based and query-based interfaces are also capable of providing information conforming to the information set. As long as the information in the information set is made available to XML applications in one way or another, the requirements of this document are satisfied".[19]

3. Creating DOMs. The W3C DOM interface does not define constructors. Thus an org.w3.dom.Document must be created through a non-DOM constructor. We have used the Apache Xerces implementations[16] which define implementations (*Impl() for all W3C interfaces), and we use the org.apache.xml.DocumentImpl() constructor to create an empty DOM. Each element within this Document must be created in its context, e.g. with Document.createElement-(tagName) and added with (say) Document.appendChild-(Node). In this way the complete document can be created from an org.xml.sax.InputSource and trapping the SAX events. Alternatively the elements can be created from legacy data and added to the DOM as appropriate.

We have decided to implement our own DOM with CMLBase as a subclass of org.w3.dom.Element. This is not completely straightforward without defined W3C DOM constructors to subclass and may involve some intermediate "glue" for different parent DOMImplementations (currently uk.co.demon.ursus.PMRDocument, etc.) such as org.apache.xml.DocumentImpl(org.xml.sax.InputSource) to create a complete DOM. Although this contains the correct elements and substructure, these are not created as subclasses, so the CMLDOM methods are not available. Elements may have to be copied to a stub set of subclassed elements.

4. Technical Considerations. In general the DOM provides all the functionality that is required for accessing or editing chemical objects. However there may be performance or implementation considerations when using a DOM such as the following:

- Building trees tends to be slow and memory-hungry.

Chart 1

```
Class Hierarchy

    class CMLBaseImpl (implements CMLBase)
        class AbstractAtomRefImpl (implements HasAtomRefs)
            class AbstractAngleImpl
                class AngleImpl (implements Angle)
                class TorsionImpl (implements Torsion)
        class AbstractBuiltinContainerImpl (implements HasBuiltinNames)
            class AbstractAtomImpl (implements AbstractAtom)
                class AtomArrayImpl (implements AtomArray)
                class AtomImpl (implements Atom)
            class AbstractBondImpl (implements AbstractBond)
                class BondArrayImpl (implements BondArray)
                class BondImpl (implements Bond)
            class CrystalImpl (implements Crystal)
        class AbstractStringValImpl
            class StringValImpl (implements StringVal)
                class FloatMatrixImpl (implements FloatMatrix)
                class NumericValImpl (implements NumericVal)
                    class FloatValImpl (implements FloatVal)
                        class FloatArrayImpl (implements FloatArray)
                    class IntegerValImpl (implements IntegerVal)
                        class IntegerArrayImpl (implements IntegerArray)
                class StringArrayImpl (implements StringArray)
        class CMLImpl (implements CML)
        class ElectronImpl (implements Electron)
        class ExecuteImpl (implements Execute)
        class FeatureImpl (implements Feature)
        class FormulaImpl (implements Formula)
        class LinkImpl (implements Link)
        class ListImpl (implements List)
        class MoleculeImpl (implements Molecule)
        class ReactionImpl (implements Reaction)
        class SequenceImpl (implements Sequence)
    class java.lang.Object
        class ArrayDelegate
        class Convention (implements ConventionCriteria)
        class jumbo.euclid.Status (implements java.lang.Cloneable)
            class jumbo.euclid.Point3
                class Coordinate3
            class jumbo.euclid.Real2
                class Coordinate2
        class java.lang.Throwable (implements java.io.Serializable)
            class java.lang.Exception
                class CMLException
        class Units
            class AngleUnits
    class CMLDocumentImpl (implements CMLDocument)
    class CMLDOMSAXBuilderImpl
        class DefaultCMLDOMSAXBuilder
    class CMLValidityImpl (implements CMLValidity)

Interface Hierarchy

    interface CMLDocument
    interface CMLValidity
    interface ConventionCriteria
    interface HasAtomRefs
        interface AbstractAtomRef (also extends CMLBase)
            interface AbstractAngle
                interface Angle
                interface Torsion
        interface AbstractBond (also extends AbstractBuiltinContainer)
            interface Bond
            interface BondArray (also extends HasBuiltinArrayNames)
    interface HasBuiltinNames
        interface AbstractBuiltinContainer (also extends CMLBase)
            interface AbstractAtom (also extends HasCount)
                interface Atom
                interface AtomArray (also extends HasBuiltinArrayNames)
            interface AbstractBond (also extends HasAtomRefs)
                interface Bond
                interface BondArray (also extends HasBuiltinArrayNames)
            interface Crystal (also extends HasBuiltinNames)
        interface Crystal (also extends AbstractBuiltinContainer)
        interface HasBuiltinArrayNames
            interface AtomArray (also extends AbstractAtom)
            interface BondArray (also extends AbstractBond)
    interface HasConvention
        interface CMLBase (also extends HasDictRef, HasId, HasTitle)
            interface AbstractAtomRef (also extends HasAtomRefs)
                interface AbstractAngle
                    interface Angle
                    interface Torsion
            interface AbstractBuiltinContainer (also extends HasBuiltinNames)
              interface AbstractAtom (also extends HasCount)
                  interface Atom
                  interface AtomArray (also extends HasBuiltinArrayNames)
              interface AbstractBond (also extends HasAtomRefs)
                  interface Bond
                  interface BondArray (also extends HasBuiltinArrayNames)
              interface Crystal (also extends HasBuiltinNames)
          interface AbstractStringVal
              interface StringVal
                  interface FloatMatrix (also extends HasUnits)
                  interface NumericVal (also extends HasUnits)
                      interface FloatVal
                          interface FloatArray (also extends HasSize)
                      interface IntegerVal
                          interface IntegerArray (also extends HasSize)
                  interface StringArray (also extends HasSize)
          interface CML
          interface Electron (also extends HasCount)
          interface Execute
          interface Feature
          interface Formula (also extends HasCount)
          interface HasExecute
              interface Molecule (also extends CMLBase, HasCount)
          interface Link
          interface List
          interface Molecule (also extends HasCount, HasExecute)
          interface Reaction
          interface Sequence
  interface HasCount
      interface AbstractAtom (also extends AbstractBuiltinContainer)
          interface Atom
          interface AtomArray (also extends HasBuiltinArrayNames)
      interface Electron (also extends CMLBase)
      interface Formula (also extends CMLBase)
      interface Molecule (also extends CMLBase, HasExecute)
  interface HasDictRef
      interface CMLBase (also extends HasConvention, HasId, HasTitle)
          interface AbstractAtomRef (also extends HasAtomRefs)
              interface AbstractAngle
                  interface Angle
                  interface Torsion
```

```
interface AbstractBuiltinContainer (also extends HasBuiltinNames)
    interface AbstractAtom (also extends HasCount)
        interface Atom
        interface AtomArray (also extends HasBuiltinArrayNames)
    interface AbstractBond (also extends HasAtomRefs)
        interface Bond
        interface BondArray (also extends HasBuiltinArrayNames)
    interface Crystal (also extends HasBuiltinNames)
interface AbstractStringVal
    interface StringVal
        interface FloatMatrix (also extends HasUnits)
        interface NumericVal (also extends HasUnits)
            interface FloatVal
                interface FloatArray (also extends HasSize)
            interface IntegerVal
                interface IntegerArray (also extends HasSize)
        interface StringArray (also extends HasSize)
interface CML
interface Electron (also extends HasCount)
interface Execute
interface Feature
interface Formula (also extends HasCount)
interface HasExecute
    interface Molecule (also extends CMLBase, HasCount)
interface Link
interface List
interface Molecule (also extends HasCount, HasExecute)
interface Reaction
      interface Sequence
interface HasId
    interface CMLBase (also extends HasConvention, HasDictRef,
    org.xmlcml.cml.HasTitle)
        interface AbstractAtomRef (also extends HasAtomRefs)
            interface AbstractAngle
                interface Angle
                interface Torsion
        interface AbstractBuiltinContainer (also extends HasBuiltinNames)
            interface AbstractAtom (also extends HasCount)
                interface Atom
                interface AtomArray (also extends HasBuiltinArrayNames)
            interface AbstractBond (also extends HasAtomRefs)
                interface Bond
                interface BondArray (also extends HasBuiltinArrayNames)
            interface Crystal (also extends HasBuiltinNames)
        interface AbstractStringVal
            interface StringVal
                interface FloatMatrix (also extends HasUnits)
                interface NumericVal (also extends HasUnits)
                    interface FloatVal
                        interface FloatArray (also extends HasSize)
                    interface IntegerVal
                        interface IntegerArray (also extends HasSize)
                interface StringArray (also extends HasSize)
        interface CML
        interface Electron (also extends HasCount)
        interface Execute
        interface Feature
        interface Formula (also extends HasCount)
        interface HasExecute
            interface Molecule (also extends CMLBase, HasCount)
        interface Link
        interface List
        interface Molecule (also extends HasCount, HasExecute)
        interface Reaction
        interface Sequence
interface HasSize
    interface FloatArray (also extends FloatVal)
    interface IntegerArray (also extends IntegerVal)
    interface StringArray (also extends StringVal)
interface HasTitle
    interface CMLBase (also extends HasConvention, HasDictRef, HasId)
        interface AbstractAtomRef (also extends HasAtomRefs)
            interface AbstractAngle
                interface Angle
                interface Torsion
        interface AbstractBuiltinContainer (also extends HasBuiltinNames)
            interface AbstractAtom (also extends HasCount)
                interface Atom
                interface AtomArray (also extends HasBuiltinArrayNames)
            interface AbstractBond (also extends HasAtomRefs)
                interface Bond
                interface BondArray (also extends HasBuiltinArrayNames)
            interface Crystal (also extends HasBuiltinNames)
        interface AbstractStringVal
            interface StringVal
                interface FloatMatrix (also extends HasUnits)
                interface NumericVal (also extends HasUnits)
                    interface FloatVal
                        interface FloatArray (also extends HasSize)
                    interface IntegerVal
                        interface IntegerArray (also extends HasSize)
                interface StringArray (also extends HasSize)
        interface CML
        interface Electron (also extends HasCount)
        interface Execute
        interface Feature
        interface Formula (also extends HasCount)
        interface HasExecute
            interface Molecule (also extends CMLBase, HasCount)
        interface Link
        interface List
        interface Molecule (also extends HasCount, HasExecute)
        interface Reaction
        interface Sequence
interface HasUnits
    interface FloatMatrix (also extends StringVal)
    interface NumericVal (also extends StringVal)
        interface FloatVal
            interface FloatArray (also extends HasSize)
        interface IntegerVal
            interface IntegerArray (also extends HasSize)
```

Chart 2

```
package org.xmlcml.cml;

public interface Atom extends AbstractAtom {

/** Vector of elements which refer to this atom */
        public Vector getAtomRefVector();

// builtins

        /** DOM: get Coordinate
        @return Coordinate2 the coordinate or null
        */
        public Coordinate2 getXY2();
        /** DOM: get 3-D Cartesian Coordinate (default Angstrom)
        @return Coordinate3 the Cartesian coordinate or null
        */
        public Coordinate3 getXYZ3();
        public Coordinate3 getXYZFract();

        public void setXY2(double x2, double y2);
        public void setXY2(Coordinate2 xy2);
        public void setXYZ3(double x3, double y3, double z3);
        public void setXYZ3(Coordinate3 xyz3);
        public void setXYZFract(double x3, double y3, double z3);
        public void setXYZFract(Coordinate3 xyzFract);

        public double getOccupancy() throws CMLException;
        public void setOccupancy(double occ);
        public Isotope getIsotope() throws CMLException;
        public void setIsotope(Isotope isotope);
        public int getFormalCharge() throws CMLException;
        public void setFormalCharge(int charge);
        public int getHydrogenCount() throws CMLException;
        public void setHydrogenCount(int hydrogenCount);
        public int getNonHydrogenCount() throws CMLException;
        public void setNonHydrogenCount(int nonHydrogenCount);
        public AtomParity getAtomParity() throws CMLException;
        public void setAtomParity(AtomParity parity);
        public String getElementType() throws CMLException;
        public void setElementType(String elementType);
        public String getResidueType() throws CMLException;
        public void setResidueType(String residueType);
        public String getResidueID() throws CMLException;
        public void setResidueID(String residueID);

        /**=======================================================*/

        /** give each atom an ID. Order is: id attribute, atomId builtin child,
then serial
        */
        public String generateId(int serial);

/** add ligand. If already known, ignore. Else look for bond including atom/ligand.
        If not present, create default bond. return Bond
        @param Atom ligand the ligand
        @return Bond the new bond
        */
        public Bond createBondToLigand(Atom ligand) throws CMLException;
/** add ligand. If already known, ignore.
        @param Atom ligand to add
        */
        public void addLigand(Atom ligand) throws CMLException;
        public Vector getLigandVector();
/** get ligand count
        @return int ligand count (0 or more)
        */
        public int getLigandCount();

/** convenience method to get element of ligand vector */
        public Atom getLigand(int i);

/** an atom may be associated with one Molecule (or none) */
        public Molecule getMolecule();

        /** get Vector of non-hydrogen ligands */
        public Vector getNonHydrogenLigandVector();

        /** convenience method to get interatomic distance */
        public Double getLength(Atom atom);
}
```

• The whole tree usually needs to be created even if only a small part is accessed.

• Traversing the whole tree may be inefficient if only a small part is required. (Since traversal can be are initiated at any node, it may be useful to build an index).

• Some data structures (especially tables) are very inefficient as trees and better handled by other methods.

5. Complete Class and Interface Hierarchy for CMLDOM 1.0. This shows the complete class and interface hiererarchy (without methods) for CMLDOM 1.0 derived from the javadoc documentation. The indentation shows the hierarchy (e.g. AngleImpl and TorsionImpl are both subclasses of AbstractAngleImpl). Since Java requires single inheritance, classes only occur once, but subclassed interfaces can extend many interfaces, so subinterfaces may occur several times (see Chart 1).

METHODS

We show the methods for a typical class (org.xmlcml.cml.Atom) in Chart 2. The first section represents DTD-required methods; the second shows some additional non-DTD convenience methods (i.e. not extending the chemical functionality, but providing more coherent code).

Supporting Information Available: CMLDOM classes. This material is available free of charge via the Internet at http://pubs.acs.org.

REFERENCES AND NOTES

(1) Murray-Rust, P.; Rzepa, H. S. *J. Chem. Inf. Comput. Sci.* **1999**, *39*, 928.
(2) An overview of these tools is available at site of the World-Wide Web consortium, see: http://www.w3c.org/.
(3) For further information, see: http://www.omg.org/.
(4) For a specification of SVG, see: http://www.w3.org/TR/2000/CR-SVG-20001102/.
(5) Knuth, D. E. *The Art of Computer Programming*, 3rd ed.; Addison-Wesley: 1997.
(6) Berners-Lee, T. *Building a Web of Trust: The Semantic Web*; Talk given at WWW9/Amsterdam, March 2000; http://www.w3.org/2000/Talks/0516-sweb-tbl/all.
(7) Jacobson, I.; Booch, G.; Rumbaugh, J. *IEEE Software* **1999**, *16*, 96. Coleman, D.; Artim, J.; Ohnjec, V.; Rivas, E.; Rumbaugh, J.; WirfsBrock, R. *Sigplan Notices* **1997**, *32*, 201−205.
(8) We have been able to transfer our CMLDOM Java bytecode to collaborators who have been easily able to use UML-based tools to represent and use the class functionality.
(9) Mathematical Markup Language (MathML) Version 2.0; http://www.w3.org/Math/ and http://www.w3.org/TR/2000/CR-MathML2−20001113/.
(10) Document Object Model (DOM) Level 2 Core Specification; http://www.w3.org/DOM/ and http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/.
(11) XML Schema; http://www.w3.org/XML/Schema.
(12) Canonical XML Version 1.0; http://www.w3.org/TR/2000/WD-xml-c14n-20000613.
(13) For an overview of XML, see: http://www.xml.org/.
(14) Murray-Rust, P.; Rzepa, H. S.; Wright, M.; Zara, S. *Chemcomm*, **2000**, 1471. Murray-Rust, P.; Rzepa, H. S.; Wright, M. *New. J. Chem*, **2001**, 618−634.
(15) For further information on this tool, see: http://java.sun.com/j2se/javadoc/.
(16) For further information on these and other tools, see: http://xml.apache.org/.
(17) For further information on these and other tools, see: http://www.xml-cml.org/.
(18) For further information on this tool, see: http://xml.apache.org/fop/.
(19) Cowan, J. http://www.w3.org/TR/xml-infoset.

CI000404A