Journal of
Applied
Crystallography

ISSN 0021-8898

Received 17 February 2011
Accepted 24 March 2011

CIFXML: a schema and toolkit for managing CIFs in XML

Nick E. Day, Peter Murray-Rust* and Simon M. Tyrrell

Unilever Centre for Molecular Informatics, Department of Chemistry, University of Cambridge, Lensfield Road, Cambridge CB2 1EW, UK. Correspondence e-mail: pm286@cam.ac.uk

CIFXML applies the XML strategies and technologies to create a general interface for processing CIF documents that conform to the CIF syntax and DDL1. Both a DTD and an XML schema for CIFs are presented. CIFs can be read, edited, validated syntactically, sorted, normalized, filtered, stored as an XML document object model, transformed and output. *CIFXOM* provides an easy way of converting CIFs to XML and *vice versa* using Java.

1. Introduction

Crystallographic information file (CIF; Hall *et al.*, 1991) is a structured document format that is in common use for the interchange of crystallographic information [the same acronym is used for the broader system of exchange protocols known as Crystallographic Information Framework (Hall & McMahon, 2005)]. It has a formal syntax, which describes 'well formed' CIFs and which is now almost completely honoured in practice. Some semantics are formalized but current usage is variable. Ontology is provided by CIF dictionaries, which in principle allow machine validation of data instances, but relatively few tools exist for semantic integration.

Several excellent CIF parsers have been developed, but most of them store the parsed information (infoset) within the memory of the program and users will need to know the internals and language of each program to extract information. Programming libraries for working with CIF files have already been described for Fortran (Hall & Bernstein, 1996) and C, or variants (Westbrook *et al.*, 1997; Hester, 2006), Python (Chang & Bourne, 1998; Edgington, 1997), and Perl (Bluhm, 2000).

The CIF standard pre-dated XML (W3C, 1997) by about a decade, and its components [datafiles, dictionaries and DDLs (dictionary definition languages)] are essentially isomorphous to the XML infrastructure [documents, schemas, XSD (XML Schema Definition)]. It is possible to represent much of the formal power of CIF DDLs in XSD. One of us (PM-R) has been through this exercise and established that when the CIF constructs are translated into XML equivalence it is possible to carry out a large amount of validation (Murray-Rust, 1998). However there are a number of constructs in CIF that cannot be trivially converted to XML and doing this explicitly is considerably more laborious than hard-coding pragmatic implicit semantics where they are essential (http://www.iucr.org/__data/iucr/cif/software/ciftbx/).

The XML community has developed many strategies and tools for semantics and ontological operations on structured documents, and we have transferred these to support CIFs by developing the XML dialect CIFXML. XML provides schema-based validation of data instances and a variety of strategies for transforming documents [Simple API (application programming interface) for XML (SAX; http://www.saxproject.org/), Document Object Model (DOM; W3C, 2005), Extensible Stylesheet Language Transformations (XSLT; W3C, 1999) and XSD (W3C, 2000)]. XML has the great benefit that it allows the infoset to be serialized independently of the program that created it. There are a very large number of tools for validating XML so that it is possible to check the structure and content of the serialized XML without knowing the domain specifics. Indeed many web browsers now contain good XML parsers which allow searching and filtering through JavaScript and related languages. Many other XML tools exist for searching, indexing and other manipulation, and the type of information is easily transformed into RDF (Resource Description Framework) and other web-friendly languages. This makes it possible to search for name–value constructs (CIFItem) in RDF-ized CIFXML.

XML serialization also allows 'round-tripping', which is an important tool for checking consistency and completeness of parsing and representation. Some information (mainly whitespace and other formatting) will be lost during the round trips but it is possible to carry out the process CIF to CIFXML to CIF to CIFXML with a high degree of stability.

There are two main strategies for processing structured documents:

(*a*) SAX. After lexical processing a document is broken into chunks, which fire events in a linear order. In XML this normally corresponds to start and end tags and contained text.

(*b*) DOM. The document is converted into a tree structure (often representable by a DTD or XML schema). The tree is held in memory and can be navigated and transformed in many ways.

SAX and DOM are complementary. SAX has the advantage of being rapid and not limited by memory. DOM preserves the context of every piece of information. In practice many XML parsers provide both strategies and use SAX to build a DOM. The use of SAX, DOM and callbacks may be unfamiliar so a brief description is given later (§7.2).

2. CIFXML

In 1995 one of us (PM-R) visited the Protein Data Bank (PDB; Berman *et al.*, 2000) in Brookhaven and worked with Professor H. J. Bernstein and colleagues on representing the emerging mmCIF (Fitzgerald *et al.*, 1996) specification in a bespoke structured markup language. Later one of the authors (PM-R) envisaged a complete suite of XML tools (Murray-Rust, 1998) that mapped onto the emerging DDLs and dictionaries, and much of this was discussed with Professors S. R. Hall and N. Spadaccini. A prototype of DDL1 and DDL2 was created in an early precursor of CIFXML, as well as a dictionary validator for the complete infrastructure of emerging DDLs and dictionaries. However, the DDL specification was still evolving at this time and even small changes gave rise to large

downstream implications in the software. The conclusion of these explorations was that building the complete infrastructure for XML representation and validation through DDLs and dictionaries was a very considerable labour and was also likely to throw up a number of semantic concerns which would have needed to have been addressed by the CIF community. At that stage, therefore, it seemed practical to hard-code the semantics into DDL1-compatible dictionaries. Since the CIF core dictionary has become relatively stable, it can be used without 'on-the-fly' validation against DDL1 (Hester, 2006).

To support CIF through standard XML methods and tools, we have now created CIFXML, an XML dialect with a corresponding XML DTD and schema. Alongside this, we have developed *CIFXOM*, a Java library for converting CIFs to valid CIFXML and *vice versa*. *CIFXOM* is based on the XML parsing strategies, and this article describes the fundamental engine for transforming CIFs into XML. The DDL-validated transformation of CIFXML documents into Chemical Markup Language (CML) will be described elsewhere (Murray-Rust *et al.*, 2011).

CIFXML currently supports the CIF syntax and DDL1-based (Hall & Cook, 1995) dictionaries [but not STAR (Cook, 1991) or DDL2 (Westbrook *et al.*, 2005), *i.e.* save frames]. It interprets any CIF as a structured document (`CIF`), which may contain the following:

(*a*) `datablocks`: these must have unique identifiers and may contain `items`, `loops` and `comments`.

(*b*) `items`: all `item` names must be unique within a `datablock`.

(*c*) `loops`: all `loops` within a `datablock` must belong to different categories (or have specific reference items), and all names in the `loop` should be unique.

(*d*) `comments`: comments can occur anywhere within a CIF where whitespace can occur. It is unclear whether comments are technically part of the content of a CIF or simply annotations for human readers only. We deprecate their use for holding information, but since they are often used for metadata we retain them in the CIFXML model.

(*e*) Whitespace: CIF elements can be separated by inline and interline whitespace, but this is not included in the CIFXML data model.

The CIF syntax allows for a number of syntactic variants such as delimiters on values or tokens used for whitespace. These are not held in the CIFXML data model so the precise lexical variant will not be recovered in round trips.

There is no formal concept of order in CIF. The data blocks, the elements within each data block and the components of a loop can be reordered without affecting the abstract data model of a CIF. According to the specification the ordering of 'rows' in a loop is not significant. However, XML supports the order of document elements and CIFXML preserves precisely all order in the input document. This allows CIFs to be 'round-tripped' (*i.e.* read into the DOM and re-output without loss). In addition, the order of the components can be canonicalized so that it is possible to compare documents with differing ordering but identical semantic content.

The CIF standard requires that data instances are valid against one or more dictionaries. In practice few tools validate CIFs against any dictionary [and we shall report elsewhere a CIFXML-based dictionary and document validation tool (Murray-Rust *et al.*, 2011)]. Certain semantics can only be applied if a dictionary is available (*e.g.* the requirement that elements in a loop must belong to the same category). These semantics are omitted from the core CIFXML model.

2.1. CIF conformance

To establish the correctness of CIFXML with respect to the schema/DTD (described below, in §4, and included in full in Appendix *A*) and to act as a CIF validator we have written a Java toolkit, *CIFXOM*. *CIFXOM* has been created to implement the CIF standard as described in the specification. We notice, however, that a small but significant fraction of CIFs do not adhere to the specification precisely. The most common deviations (which probably arise from using normal text editors rather than CIF-aware ones) are

(*a*) incorrect use of delimiters (*e.g.* assuming that end-of-line closes quotes),

(*b*) duplication of items,

(*c*) duplicate datablock names,

(*d*) improper insertion of 'comments' (sometimes apparently added by technical editors) that do not start with '#',

(*e*) illegal characters (especially non-printing characters).

CIFXOM provides some optional heuristics to attempt recovery from these, but cannot, of course, guarantee that the result is what was intended. We note that the proportion of these errors is declining, presumably as a result of the greater use of *checkCIF* (mandated by some publishers), conformance in software and the greater familiarity with CIF in the editing processes. Until relatively recently, few if any CIF-aware editing tools existed and manual editing was required for the majority of the CIF creation process. The Cambridge Crystallographic Data Centre (CCDC) provides a free (for individual research and teaching use) tool (*enCIFer*; CCDC, 2004) which allows even inexperienced users to generate syntactically correct CIFs. Another aid for pre-publication validation and formatting of CIFs is *publCIF* (Westrip, 2010), available from the International Union of Crystallography web site (http://www.iucr.org/resources/cif/software/).

Table 1
Elements in the XML schema and DTD.

Element	Sub-elements	Attribute names
`cell`	`comment`, `datablock`	`su`
`cif`	–	–
`comment`	–	–
`datablock`	`comment`, `item`, `loop`	`id`
`id`	–	–
`item`	–	`name`, `su`, `numericValue`, `dataType`
`loop`	`row`	`names`
`row`	`cell`	–

3. *CIFXOM* functionality

CIFXOM supports the following operations:

(*a*) Complete syntactic validation of CIF documents.

(*b*) Dictionary-free semantic validation against the CIF standard.

(*c*) Conversion of escaped characters to their Unicode equivalents.

(*d*) Reporting of errors and warnings with original line numbers. Further processing continues after warnings and we attempt optional recovery from some errors.

(*e*) Optional parsing of numbers with standard uncertainty fields [*e.g.* 123.45(6)].

(*f*) Choice of DOM or SAX strategies and choice of parsers.

(*g*) Creation of a CIFXML object from CIF or XML.

(*h*) Normalization of document structure.

(*i*) Canonicalization of document structure.

(*j*) Optional sorting of part or whole document.

(*k*) Identification of differences between data models for two CIFs (*i.e.* independent of syntax and ordering).

(*l*) Output as XML, HTML or CIF for round-tripping.

4. Representation of CIF documents in XML

The DTD to which the XML serialization of CIFs must conform is included in full in Appendix *A*, as is its XML schema representation. The elements are listed and described in Table 1.

Table 2
A comparison of CIF and CIFXML representation.

CIF structure	Equivalent CIFXML structure
	`<cif>`
`data_I`	`<datablock id = "I">`
`_audit_creation_method SHELXL97`	`<item name = "_audit_creation_method">SHELXL97</item>`
`_chemical_formula_sum`	`<item name = "_chemical_formula_sum">`
`'C93 H84 Cl3 Co Fe N8 O2'`	`C93 H84 Cl3 Co Fe N8 O2</item>`
`_chemical_formula_weight 1566.81`	`<item name = "_chemical_formula_weight">1566.81</item>`
`_symmetry_cell_setting 'Triclinic'`	`<item name = "_symmetry_cell_setting">Triclinic</item>`
`_symmetry_space_group_name_H-M 'P -1'`	`<item name = "_symmetry_space_group_name_h-m">P -1</item>`
`loop_`	`<loop names = "_symmetry_equiv_pos_as_xyz">`
`_symmetry_equiv_pos_as_xyz`	
`'x, y, z'`	`<row>`
	`<cell>x, y, z</cell>`
	`</row>`
`'-x, -y, -z'`	`<row>`
	`<cell>-x, -y, -z</cell>`
	`</row>`
	`</loop>`
`_cell_length_a 13.8463(3)`	`<item name = "_cell_length_a">13.8463(3)</item>`
`_cell_length_b 16.8164(5)`	`<item name = "_cell_length_b">16.8164(5)</item>`
`_cell_length_c 17.9072(6)`	`<item name = "_cell_length_c">17.9072(6)</item>`
`_cell_angle_alpha 93.7800(10)`	`<item name = "_cell_angle_alpha">93.7800(10)</item>`
`_cell_angle_beta 111.1430(10)`	`<item name = "_cell_angle_beta">111.1430(10)</item>`
`_cell_angle_gamma 97.4630(10)`	`<item name = "_cell_angle_gamma">97.4630(10)</item>`
`_cell_volume 3827.19(19)`	`<item name = "_cell_volume">3827.19(19)</item>`
`_cell_formula_units_Z 2`	`<item name = "_cell_formula_units_z">2</item>`
`_cell_measurement_temperature 110(2)`	`<item name = "_cell_measurement_temperature">110(2)</item>`
	`</datablock>`
	`</cif>`

Using this DTD/schema, we show in Table 2 how a fragment of a typical CIF is translated.

An alternative syntax for the numeric fields, which avoids the problems of parsing suffixed brackets, is exemplified by the following:

```
<item name="_cell_length_a" su="0.0003"
    numericValue="13.8463" dataType="numb">13.8463(3)</item>
<item name="_cell_length_b" su="0.0005"
    numericValue="16.8164" dataType="numb">16.8164(5)</item>
<item name="_cell_length_c" su="0.0006"
    numericValue="17.9072" dataType="numb">17.9072(6)</item>
<item name="_cell_angle_alpha" su="0.0010"
    numericValue="93.7800" dataType="numb">93.7800(10)</item>
<item name="_cell_angle_beta" su="0.0010"
    numericValue="111.1430" dataType="numb">111.1430(10)</item>
<item name="_cell_angle_gamma" su="0.0010"
    numericValue="97.4630" dataType="numb">97.4630(10)</item>
<item name="_cell_volume" su="0.19"
    numericValue="3827.19" dataType="numb">3827.19(19)</item>
<item name="_cell_formula_units_z">2</item>
<item name="_cell_measurement_temperature" su="2.0"
    numericValue="110" dataType="numb">110(2)</item>
```

5. *CIFXOM* architecture

CIFXOM is a single package based closely on the SAX model. We have used the simple and elegant XOM (http://www.xom.nu/) model rather than the overly engineered and difficult W3C DOM model. *CIFXOM* contains the following main classes, most of whose functionality is obvious from the name or the position in the class hierarchy. The CIF parsing uses a SAX-like model where events cause callbacks to the content or error handlers.

(*a*) `AbstractBlock.java`
(*b*) `AbstractTextElement.java`
(*c*) `AbstractValueElement.java`
(*d*) `CIFComment.java`
(*e*) `CIFContentHandler.java`
(*f*) `CIFDataBlock.java`
(*g*) `CIFElement.java`
(*h*) `CIFErrorHandler.java`
(*i*) `CIFException.java`
(*j*) `CIFItem.java`
(*k*) `CIFLoop.java`
(*l*) `CIFParser.java`
(*m*) `CIFRow.java`
(*n*) `CIFSaveFrame.java`
(*o*) `CIFTableCell.java`
(*p*) `DOMBuilderContentHandler.java`
(*q*) `DefaultContentHandler.java`
(*r*) `DefaultErrorHandler.java`

The inheritance hierarchy of the main *CIFXOM* concrete classes is shown in Fig. 1. All *CIFXOM* elements are descendants of the XOM `Element` class.

The base class is `CIFElement`, which defines a basic API for processes common to all subclasses.

(i) `String toCIFString()`
This returns the `CIFElement` as a CIF-formatted string.

(ii) `void writeXML (Writerw) throws IOException`
This will output the `CIFElement` and all of its children in an XML format.

(iii) `void writeHTML (Writerw) throws IOException`
This will output the `CIFElement` and all of its children in an HTML format with lists converted into HTML tables.

(iv) `void writeCIF (Writerw) throws IOException`
This will output the `CIFElement` and all of its children in CIF format, thus showing that *CIFXOM* is a lossless library. It uses the `toCIFString()` method described above.

(v) `void normalize()`
This will attempt to remove any lexical variants.

(vi) `void canonicalize()`
Within a CIF file the order of the `datablocks`, `items` and `loops` (including the row/column ordering) are all arbitrary. This will reorganize the order of the various `CIFElements` within a `CIFDocument` into a lexical order. The default behaviour of `canonicalize()` is to apply the following heuristics during its reordering:

(1) `CIFItems` occur lexically before `CIFLoops`,

(2) `CIFItems` are sorted alphabetically by name,

(3) the columns of each `CIFloop` are sorted alphabetically by namelist, then the rows are sorted upon their lexical ordering,

(4) the `CIFLoops` are sorted alphabetically using the name of their first column.

(vii) `void processSu(boolean b)`
This determines whether numeric variables with standard uncertainties in brackets should be parsed and analysed.

As a further illustration, an example of the canonicalization algorithm for a small set of CIF data is given in Fig. 2.

6. Installing *CIFXOM*

CIFXOM requires Java 1.5 or higher (http://www.javasoft.com) and is available under Artistic License 2.0 (http://www.opensource.org/

Table 3
Some example methods of the `CIFItem` class API.

Declaration	Description
`public void setItemName(String name) throws CIFException;`	Set the name for a data item: Normally used when building CIFXML Data names should never be reset Implementers may check the value of a name or whether it violates any CIF syntax or dictionary restrictions Parameters: name (should be compliant with CIF syntax) Throws: `CIFException` syntax violation or ontology/dictionary violation
`public void setItemValue(String value) throws CIFException;`	Set the value for a data item: Normally used when building CIFXML Implementers may check the value to see whether it violates any CIF syntax or dictionary restrictions Parameters: value (should be compliant with CIF syntax); no quotes are permitted unless part of the value Throws: `CIFException` syntax violation or ontology/dictionary violation
`public String getItemName();`	Get the name for a data item: Returns: the name (should never be null)
`public String getItemValue();`	Get the value for a data item: Returns: the value
`public Double getSU();`	Get the standard uncertainty for a data item: CIF parsers should ensure that if s.u. is non-blank then the data value should not contain a bracketed s.u. Returns: the standard uncertainty (null if not present)

licenses/artistic-license-2.0) from the CML project at Sourceforge (http://sourceforge.net/projects/cml/). The latest distribution can be downloaded as a jar file (http://sourceforge.net/projects/cml/), or the source code can be downloaded from the Subversion/CVS repositories (http://sourceforge.net/projects/cml/develop) using an appropriate client. To build the source code, Maven 2.0 (http://maven.apache.org/) is recommended. Simple examples, expected output and unit tests can be found in both the distribution and the code repository.

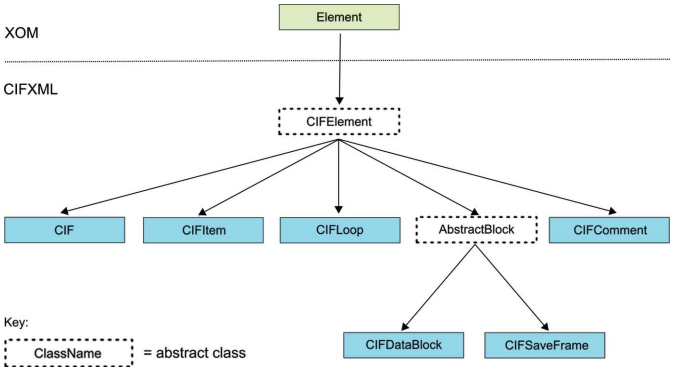

Figure 1
The *CIFXOM* inheritance hierarchy (`CIFSaveFrame` is reserved for expansion).

Before canonicalization	After canonicalization

`data_xY`
 	
`_ITEM2 1.2`
 	
`loop`
`_col2 _col1`
`99 4`
`101 3`
 	
`_item1 1.1`
 	
`loop`
`_zzz _aaa`
`1 z`
`99 q`
	`data_xY`
	
	`_item1 1.1`
	`_item2 1.2`
	
	`loop`
	`_aaa _zzz`
	`z 1`
	`q 99`
	
	`loop`
	`_col1 _col2`
	`4 99`
	`3 101`

Figure 2
Example of applying the canonicalization algorithm.

7. Using *CIFXOM*

CIFXOM is a toolkit and can be used for many purposes. A few standard tasks have been programmed and these will also be valuable for understanding how to use the toolkit. All classes are fully documented and are thus supported by Javadoc (http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html), which is recommended as a useful ancillary tool.

As all *CIFXOM* elements are subclassed from the XOM `Element` class, *CIFXOM* uses many XML functions from the XOM library. Therefore, application builders may find it useful to refer to the documentation and tutorials about XOM (http://www.xom.nu/).

7.1. Examples of the API

Each of the CIF classes has an API to facilitate the programmatic adding, removing, setting and getting of its particular data fields. For example, some of the methods of `CIFItem` are shown in Table 3.

7.2. Parsing and callbacks

CIFXOM has a default parsing system which can be subclassed should a different parsing mechanism be needed. This allows the implementer or user to choose between parsers (including at runtime), perhaps on the basis of speed or conformance. In practice most programmers will use the default.

The SAX strategy is that a parser provides callbacks when lexical/document events are fired. This means that the user delegates the parsing process to a parser and only regains control after a complete parse (unless exceptions are thrown). The user provides callbacks to trap the events so that any that are not required can be ignored.

The following code is an excerpt from the `readToken` method of the `CIFParser` class, which shows a callback to the `CIFContentHandler` (`contentHandler` in the code) to add a `CIFItem` (`item`) to the current instance of a `CIFDataBlock` (`this`). If there is an error during this method call, there is a callback to the `CIFErrorHandler` (`errorHandler`) to provide the error message.

```
1: ParserMessage m = contentHandler.addItem(item);
2: if (m != null) {
3:     errorHandler.error(m.getMessage(), this);
4: }
```

7.3. Example use of the `CIFParser` class

Code to read a CIF into CIFXML, canonicalize it and then write out the CIFXML is included in full in Appendix *B*.

7.4. A simple CIF editor

A simple use case involves reading a CIF into CIFXML and manipulating it through DOM-like calls, thus providing some of the features of a simple editing system. After creating the CIF, the process iterates over the `datablock`s and, for instance, manipulates the `_cell_measurement_temp` item. In the example provided, it will either add a new item or change the value of the current one. The code is included in full in Appendix *C*.

8. Deployment

CIFXOM has already been implemented in the following:

(*a*) The CrystalEye (http://wwmm.ch.cam.ac.uk/crystaleye/) web site, a crystallographic repository containing over 120 000 CIF files, all of which have been processed by *CIFXOM* (parsing, manipulation of the CIF data structure and input for conversion into CML). This has exposed CIFXML to CIFs from a wide range of laboratories with varying degrees of conformance to the exact standard.

(*b*) The SPECTRa (Downing *et al.*, 2008) and SPECTRa-T (Downing *et al.*, 2010) projects, in which *CIFXOM* was similarly implemented as a component of repository software implemented at the University of Cambridge, Imperial College London and the University of Southampton.

APPENDIX *A*
DTD and XSD schema

The DTD to which the XML serialization of CIFs must conform is as follows:

```
<!DOCTYPE cif [
<!ELEMENT cif (datablock | comment)*>
<!ELEMENT datablock (comment | item | loop)*>
<!ATTLIST datablock
          id CDATA #REQUIRED>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT item (#PCDATA)>
<!ATTLIST item
          name CDATA #REQUIRED
          su CDATA #IMPLIED
          numericValue CDATA #IMPLIED
          dataType CDATA #IMPLIED>
<!ELEMENT loop (row+)>
<!ATTLIST loop
          names CDATA #REQUIRED>
<!ELEMENT row (cell+)>
<!ELEMENT cell (#PCDATA)>
<!ATTLIST cell
          su CDATA #IMPLIED>
]>
```

This DTD can also be expressed as an XML schema, as in the following:

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
  <xs:element name='cell'>
    <xs:complexType mixed='true'>
    <xs:attribute name='su'/>
    </xs:complexType>
  </xs:element>
  <xs:element name='cif'>
    <xs:complexType>
      <xs:choice minOccurs='0' maxOccurs='unbounded'>
        <xs:element ref='datablock'/>
        <xs:element ref='comment'/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name='comment'>
    <xs:complexType mixed='true'>
    </xs:complexType>
  </xs:element>
  <xs:element name='datablock'>
    <xs:complexType>
      <xs:choice minOccurs='0' maxOccurs='unbounded'>
        <xs:element ref='comment'/>
        <xs:element ref='item'/>
        <xs:element ref='loop'/>
      </xs:choice>
      <xs:attribute name='id' use='required'/>
    </xs:complexType>
  </xs:element>
  <xs:element name='item'>
    <xs:complexType mixed='true'>
      <xs:attribute name='name' use='required'/>
      <xs:attribute name='su'/>
      <xs:attribute name='numericValue'/>
      <xs:attribute name='dataType'/>
    </xs:complexType>
  </xs:element>
  <xs:element name='loop'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='row' maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:attribute name='names' use='required'/>
    </xs:complexType>
  </xs:element>
  <xs:element name='row'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='cell' maxOccurs='unbounded'/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

APPENDIX *B*
Example use of the `CIFParser` class

The following code will read a CIF into CIFXML, canonicalize it and then write out the CIFXML:

```
 1: package sandbox;
 2:
 3: import java.io.IOException;
 4: import nu.xom.Document;
 5: import org.xmlcml.cif.CIF;
 6: import org.xmlcml.cif.CIFException;
 7: import org.xmlcml.cif.CIFParser;
 8:
 9: public class ParseIntoCifDom {
10:    public static void main(String[] args) {
11:       String filename = "C:/path/to/your/cif/file.cif";
12:       try {
13:          CIFParser parser = new CIFParser();
14:          // Set 'skip errors' option.  Applies heuristics
15:          // telling the parser to fix or skip CIF
16:          // sections with recoverable errors.
17:          parser.setSkipErrors(true);
18:          // Set 'skip header' option.  Tells the parser to
19:          // skip any comments that occur before the first
20:          // datablock.
21:          parser.setSkipHeader(true);
22:          // Parse CIF into XML document using XOM
23:          Document doc = parser.parse(filename);
24:          // Root element is always a CIF object
25:          CIF cif = (CIF) doc.getRootElement();
26:          // Apply canonicalization algorithm to CIF object.
27:          cif.canonicalize();
28:          String outfile = "C:/path/to/write/to.xml";
29:          // Write CIFXML to path specified as 'outfile'
30:          FileWriter fw = new FileWriter(outfile);
31:          cif.writeXML(fw);
32:          fw.close();
33:       } catch (Exception e) {
34:          System.err.println("Error processing CIF file: "
35:                      +filename);
36:       }
37:    }
38: }
```

APPENDIX *C*
A simple CIF editor

The following code manipulates the `_cell_measurement_temp` item, either adding a new item or changing the value of the current one.

```
 1: String file = "C:/path/to/your/cif/file.cif";
 2: try {
 3:    String tempItemName = "_cell_measurement_temperature";
 4:    // values for the new temperature, standard uncertainty
 5:    // and the number of decimal places to be used.
 6:    double newTemp = 205.0;
 7:    double newSu = 1.0;
 8:    int dps = 1;
 9:    // parse the CIF file and get the root CIF element
10:    CIF cif = (CIF) new CIFParser().parse(file)
11:          .getRootElement();
12:    // for each datablock in the CIF
13:    for (CIFDataBlock block : cif.getDataBlockList()) {
14:       // try to find the cell measurement temperature item
15:       CIFItem temp = block.getChildItem(tempItemName);
16:       if (temp == null) {
17:          // if one doesn't exist then create a new one with
18:          // the values above, and add to the datablock
19:          temp = new CIFItem(tempItemName, newTemp,
20:             newSu, dps);
21:          block.appendChild(temp);
22:       } else {
23:          // if one does exist, then change its values to
24:          // those specified above
25:          temp.setValueAndSu(newTemp, newSu, dps);
26:       }
27:    }
28:    // write the CIF back out over the old file
29:    FileWriter fw = new FileWriter(file);
30:    cif.writeCIF(fw);
31:    fw.close();
32: } catch (Exception e) {
33:    System.err.println("Error processing CIF file: "
34:                +filename);
35: }
```

We thank the DTI/EPSRC for support under the UK eScience program. NED thanks the EPSRC for a studentship. The invaluable assistance of Dr Charlotte Bolton in the preparation of this manuscript is acknowledged.

References

Bluhm, W. (2000). *STAR* (*CIF*) *Parser*, http://pdb.sdsc.edu/STAR/index.html.
Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., Weissig, H., Shindyalov, I. N. & Bourne, P. E. (2000). *Nucleic Acids Res.* **28**, 235–242.
CCDC (2004). *enCIFer*, http://www.ccdc.cam.ac.uk/free_services/encifer/.
Chang, W. & Bourne, P. E. (1998). *J. Appl. Cryst.* **31**, 505–509.
Cook, A. P. F. (1991). *Implementing SMD in STAR: Dictionary Definition Language*. ORAC Ltd, Leeds, UK.
Downing, J., Harvey, M. J., Morgan, P. B., Murray-Rust, P., Rzepa, H. S., Stewart, D. C., Tonge, A. P. & Townsend, J. A. (2010). *J. Chem. Inf. Model.* **50**, 251–261.
Downing, J., Murray-Rust, P., Tonge, A. P., Morgan, P., Rzepa, H. S., Cotterill, F., Day, N. & Harvey, M. J. (2008). *J. Chem. Inf. Model.* **48**, 1571–1581.
Edgington, P. R. (1997). *HICCuP: High-Integrity CIF Checking Using Python*. Cambridge Crystallographic Data Centre, UK.
Fitzgerald, P. M. D., Berman, H. M., Bourne, P. E., McMahon, B., Watenpaugh, K. D. & Westbrook, J. (1996). *Acta Cryst.* A**52**(Suppl), MSWK.CF.06.
Hall, S. R., Allen, F. H. & Brown, I. D. (1991). *Acta Cryst.* A**47**, 655–685.
Hall, S. R. & Bernstein, H. J. (1996). *J. Appl. Cryst.* **29**, 598–603.
Hall, S. R. & Cook, A. P. F. (1995). *J. Chem. Inf. Comput. Sci.* **35**, 819–825.
Hall, S. R. & McMahon, B. (2005). Editors. *International Tables for Crystallography*, Volume G, *Definition and Exchange of Crystallographic Data.* Heidelberg: Springer.
Hester, J. R. (2006). *J. Appl. Cryst.* **39**, 621–625.
Murray-Rust, P. (1998). *Acta Cryst.* D**54**, 1065–1070.
Murray-Rust, P., Adams, S. E., Day, N. E., Downing, J., England, N. W. & Townsend, J. A. (2011). In preparation.
W3C (1997). *XML Core Working Group Public Page*, http://www.w3.org/XML/Core/.

W3C (1999). *XSL Transformations* (*XSLT*), http://www.w3.org/TR/xslt.
W3C (2000). *XML Schema*, http://www.w3.org/XML/Schema.
W3C (2005). *Document Object Model* (*DOM*), http://www.w3.org/DOM/.
Westbrook, J. D., Berman, H. & Hall, S. R. (2005). *International Tables for Crystallography*, Volume G, *Definition and Exchange of Crystallographic Data*, ch. 2.6, edited by S. R. Hall & B. McMahon. Heidelberg: Springer.
Westbrook, J. D., Hsieh, S.-H. & Fitzgerald, P. M. D. (1997). *J. Appl. Cryst.* **30**, 79–83.
Westrip, S. P. (2010). *J. Appl. Cryst.* **43**, 920–925.