Application and Uses of CML within the *e*Minerals project

Toby O. H. White[1], Peter Murray-Rust[2,] Phil A. Couch[3], Rik P. Tyer[2] , Richard P. Bruin[1],

Ilian T. Todorov[3], Dan J. Wilson[4] , Martin T. Dove[1], Kat F. Austen[1] , Steve C. Parker[5]

[1]Department of Earth Sciences, Downing Street, Cambridge. CB2 3EQ
[2]University Chemical Laboratory, Lensfield Road, Cambridge. CB2 1EW
[3]CCLRC Daresbury Laboratory, Daresbury, Warrington. WA4 4AD
[4]Institut für Mineralogie und Kristallographie. J. W. Goethe-Universität, Frankfurt.
[5]Department of Chemistry, University of Bath. BA2 7AY

Abstract

Within the *e*Minerals project we have been making increasing use of CML (the Chemical Markup Language) for the representation of our scientific data. The original motivation was primarily to aid in data interoperability between existing simulation codes, and successful results of CML-mediated inter-code communication are shown. In addition, though, we have discovered several other areas where XML technologies have been invaluable in developing an escientific virtual organization, and benefiting collaboration. These areas are explored, and we show a number of tools which we have constructed. In particular, we demonstrate 1) a general library, FoX for allowing Fortran programs to interact with XML data, 2) a general CML viewing tool, ccViz, and 3) an XPath abstraction layer, AgentX.

1. Introduction

1.1 Introduction to *e*Minerals

The *e*Minerals project is a NERC testbed escience project. Our remit is to study environmentally relevant problems, using molecular-scale modelling techniques, while developing and using escience technologies which directly improve the quality of research we produce.

To this end, the *e*Minerals project encompasses, on the scientific side, researchers from a number of UK universities and research institutions, who represent a broad section of the theoretical and computational environmental science community. These researchers have expertise on a wide variety of modelling codes. Indeed, we have within our team key members of the development teams for several widely used simulation codes (for example, SIESTA has over 1000 users world-wide, and DL_POLY has several thousand.)

On the escience front, therefore, our challenge is to harness this rich expertise, and facilitate collaboration and cross-fertilization between these overlapping areas of science. This paper will show how XML technologies have enabled that, and highlight a number of tools that have resulted from the project.

1.2 Introduction to CML

CML (Chemical Markup Language)[1] was in fact the first example of a full XML language and application. Although initially designed around specifically chemical vocabularies, it has proved very flexible, and more than able to take the additional semantic burden of computational atomic molecular and molecular physics codes.

1.3 *e*Minerals CML background

The eMinerals project has been working with CML for several years now, and we have reported on progress in previous years[2,3]. Our experience has been wholly positive, and CML is playing an increasingly important rôle throughout the project, above and beyond the niches we initially envisaged it filling.

2. CML in *e*Minerals

As mentioned in the introduction, the *e*Minerals project includes scientists from a range of different backgrounds, who work with a wide range of codes. For example, two widely used codes on the project are SIESTA[4], a linear-scaling electronic structure code; and DL_POLY-3[5], a classical molecular dynamics code which uses empirical potentials. In addition, we also use a number of other simulation codes (amongst which are OSSIA[6], RMCProfile [7], METADISE[8]) all written in Fortran.

All of these codes accept and emit chemical, physical, and numerical data, but each uses its own input and output formats. This presents a number of challenges when scientists from different backgrounds, familiar with different codes, have to collaborate:

- when trying to exchange data, format translation and data conversion steps are necessary before different codes can understand the data.
- translation at the human level is necessary, since a scientist familiar with the look and feel of DL_POLY output may not understand SIESTA output, nor know where to look for equivalent data.

Both of these problems can be addressed using XML technologies, and we expand upon this below.

A problem, by no means unique to this project, is that all of our scientific simulation codes are written in Fortran, of varying ages and styles. There are a number of potential approaches to interfacing Fortran and XML; the approach we have adopted is to write an extensive library, in pure Fortran, which exposes XML interfaces in a Fortran idiom. Its design and implementation are briefly explained in section 3.

Having succeeded in making our Fortran codes speak XML, we have found three areas in particular where XML output has been useful. These are briefly explained below, and the tools and methods we have developed are explained in sections 4, 5, and 6.

2.1 Data transfer between codes

When considering the rôle of XML within a project involved in computational science, with multiple simulation codes in use, the temptation is first to think about its use in terms of a common file format which would allow easier data interchange between codes; and indeed that is the perspective from which the *e*Minerals project first approached XML.

The potential uses of the ability to easily share data between simulation codes are manifold. For example, as mentioned previously, we have multiple codes available, and they are capable of doing conceptually similar things, but using different techniques. We might wish to study the same system using both empirical potentials (with DL_POLY) and quantum mechanical DFT (with SIESTA). This would enable us to gain a better appreciation of the different approximations inherent in each method, and better understand the system.

This complementary use of two codes would be made much easier if we could use identical input files for both codes, rather than having to generate multiple representations of the same data. Without any such ability, extra work is required, and there is the potential for errors creeping in as we move between representations.

Furthermore, we might wish to use the output of one code as the input to another. We might wish to extract a small piece of the output of a low accuracy simulation, and study it in much greater depth with our more precise code. Conversely, we might want to take the output of a highly accurate initial calculation, and feed it into a low accuracy code to get more results.

In either of these cases, our workload would be greatly reduced if we could simply pass output directly from one code to another, without concerning ourselves with conversion of representations.

However, the route to this lofty though apparently simple goal of data interoperability is beset by a multitude of complicating factors. We have made much progress towards it, but due to its complications, we have described it last, in section 6.

However, along the way we have discovered several other areas where XML has aided us in our role as an escience testbed project, and these we shall describe first, and expand in sections 4 and 5.

2.2 Data visualization and analysis

One of the major features of XML is the ease with which it may be parsed. Writing an XML parser is by no means trivial, but for almost all languages and toolkits in current use, the work has already been done. Thus, to write a tool which takes input from an XML file, a developer need only interact with an in-memory representation of the XML.

When a computational scientific simulation code is executed, it will produce output in some format which has its origins in a more-or-less humanly readable textual form. Often, though, accretion of output will have rendered it less comprehensible - and in any case a long simulation may well result in hundreds of megabytes of output, which can certainly not be easily browsed by eye.

Output from calculations serves two purposes. First, it enables the scientist to follow the calculation's progress as the job is running, and, once it has finished, to ensure that it has done so in the proper manner, and without errors.

Secondly, it is rare that the scientist is only interested in the fact that the job has finished. Usually, they want to extract further, usually numerical, data from the output, and explore trends and correlations.

These two aims are rarely in accord and this results in output formats having little sensible logical structure. Thus, for some purposes, the scientist will scan the output by eye, while for others, tools must be written to parse the format, and extract and perhaps graphically display relevant data.

XML formats are optimized for ease of computational processing at the expense of human readability. Thus if the simulation output is in XML, scanning by eye becomes infeasible, but processing and visualization tools may be written with much greater ease. If the XML format is common these tools may be of wide applicability. We detail the construction of such tools below, in section 4.

2.3 Data management

In addition to efforts towards data interoperability, and data visualization, there is a third area where we have found XML invaluable, that of data management.

The use of grid computing has brought about an explosion in the volume of data generated by the individual researcher. eScience tools have enabled the generation of multiple jobs to be run, allocation of jobs to available computational resources, recovery of job output, and efficient distributed storage of the resultant data. We have addressed all of these to some extent within our project[9] and we are by no means unique in this.

However, given this large volume of data, categorization is extremely important to facilitate retrieval in both the short term when knowledge on the purpose and context of the data is still to hand and long term, when it may not be. Of equal importance is that when collaborating with geographically disparate colleagues, the data must be comprehensible by both the original investigator who will have some notion of the data's context, and by other investigators who may not.

This is related of course to the perennial problem of metadata, to which there are many approaches. The solution we have come up with in the *e*Minerals project depends heavily on the fact that our data is largely held as XML, which makes it much easier to automatically parse the data to retrieve useful data and metadata by which to index it.

The tools and techniques used are explained further in section 5.

3. Fortran and XML

It was mentioned in the Introduction that we were faced with the problem of somehow interfacing our extensive library of Fortran codes with the XML technologies we wished to take advantage of. There are a number of potential approaches to this issue.

We could write a series of translation scripts, or services, using some XML-aware language, which would convert between XML and whatever existing formats our codes understood. This however requires a multiplication of components, and increases the fragility of our systems.

Alternatively, we could write wrappers for all our codes, again in some XML-aware language, which hid the details of our legacy I/O behind an XML interface. Again though, this is a potentially fragile approach, and requires additional setup of the wrapper wherever we wish to run our codes.

A third potential solution is to reverse the problem, and use Fortran to wrap an existing XML library written in another language, probably C, so that the codes might directly call XML APIs from Fortran, and our workflows ignore the legacy I/O. However, Fortran cross-language compilation is fraught with difficulties, and in addition we would need to ensure that our C library was available and compiled on all platforms where we wished to run.

The solution that we have adopted is to write, from scratch, a full XML I/O library in Fortran, and then allow our Fortran codes to use that.

One of the major advantages of Fortran, and one of the reasons why it is in continued use in the scientific world, is that compilers exist for every platform, and Fortran code is extremely portable across the 9 or 10 compilers, and 7 or 8 hardware platforms, which are commonly available. There is no XML-aware language which is as portable, and cross-language compilation over that number of potential systems is a painful process. So, although writing the whole library from scratch in Fortran does involve more work than leveraging an existing solution, it achieves maximum portability, and means we are not restricted at all upon where our XML-aware codes may run.

The library we have written is called FoX (Fortran XML)[10] and an earlier version (named xmlf90) was described in [2]. In its current incarnation, it consists of four modules, which may be used together or independently. Two of them provide APIs for input APIs, and two for output.

On the input side, the modules provide APIs for event-driven processing, SAX 2[11], and for DOM Core level 2[12]. On the output side, there is a general-purpose XML writing library; built on top of which is a highly abstracted CML output layer. It is written entirely in standard Fortran 95.

The SAX and DOM portions of the library contain all calls provided by those APIs, modified slightly for the peculiarities of Fortran. An overview of their initial design and capabilities may be seen in [2] although the current version of FoX has significantly advanced, not least in now having full XML Namespaces[13] support. The two output modules, wxml and wcml are described here.

3.1 wxml

Clearly XML can be output simply by a series of print statements, in any language. However, XML is a tightly constrained language, and simple print statements are firstly very prone to errors, and secondly, make correct nesting of XML across a document impossible for anything but the most simple of applications. The requirement for good XML output libraries has long been recognized.

However, for most languages, simple XML output libraries rarely exist, or tend to be second-class citizens. More usually, XML output tends to be a simple addendum to a DOM library. If so, a method is provided which serializes the in-memory DOM. Thus as long as your data structures are held as a tree in memory, XML output is trivial.

Indeed, FoX supplies such a method with its DOM implementation. For applications that can easily be written around a tree-like data structure, this is fine. However, for nearly all existing Fortran applications, this is of no use whatsoever; Fortran is not a language designed with tree-like data-structures in mind, and in any case most Fortran developers are unfamiliar with anything more complicated than arrays. Forcing all data in a simulation code to be held within a DOM-like model would be entirely unnatural.

Furthermore, simulation codes often produce extremely large quantities of data, and may do so over extended periods of time. It would be foolish to keep this data all in memory for the entirety of the simulation. Not only would it be a vast waste of

memory, but since one would only be able serialize once the run is over and all the data complete, if the run were interrupted, all data would be lost. In addition, it is very important that one be able to keep track of the simulation as it progresses by observing its output, which requires progressive output, rather than all-in-one serialization.

Much of this could be avoided, of course, by occasional serialization of a changing DOM tree; or even by temporary conversion of the large Fortran arrays to a DOM and then step-by-step output; or, indeed, by the method we use, which is direct XML serialization of the Fortran data.

Thus, the FoX XML output layer (termed *wxml* here) is a collection of subroutines which generate XML output - tags, attributes and text - from Fortran data. The library is extensive, and allows the output of all structures described by the XML 1.1 & XML Namespaces specifications[13,14], although for most purposes only a few are necessary. XML output is obtained by successive calls to functions named **`xmlNewElement`**, **`xmlAddAttribute`**, and **`xmlEndElement`**. Furthermore, there are a series of additional function calls to directly translate native Fortran data structures to appropriate collections of XML structures. State is kept by the library to ensure well-formedness throughout the document.

Thus *wxml* enables rapid and easy production of well-formed XML documents, providing the user has a good grasp of the XML they want to produce.

3.2 wcml

However, the impetus for the creation of this library was the desire to graft XML output onto existing codes; specifically CML output. *wxml* is insufficient for this for a number of reasons:

- the developers of traditional Fortran simulation codes are, by and large, ignorant of XML, and entirely content to stay that way.
- when adapting a code to output XML, it is important that any changes made not obscure the natural flow of the code.
- it is desirable to write specialized functions to output CML elements, and common groupings thereof, directly (if for no other reason than to avoid misspellings of element and attribute names in source code.)
- further, it is useful to maintain a "house style" for the CML.

Thus, for example, consider a code that wishes to output the coordinates of a molecular configuration. This is represented in CML by a 3-level hierarchy of various tags, with around 20 different optional attributes, and 4 different ways of specifying the coordinates themselves. The average simulation code developer does not wish to concern themselves with the minutiae of these details; they certainly do not wish to clutter up the code with hundreds of XML output statements.

In fact, were this output to be encoded in the source as a lengthy series of loops over *wxml* calls, it is almost certain that even if it were written correctly initially, as the code continued to be developed by XML-unaware developers, it would eventually break.

Therefore, FoX provides an interface where these details are hidden from view:

```
call cmlAddMolecule(xf=xmlFile,
       coords=array_of_coords,
       elems=array_of_elements)
```

which outputs the following chunk of CML:

```
<molecule>
  <atomArray>
    <atom xyz3="0.1 0.1 0.1"
       elementType="H"/>
    ....
  </atomArray>
</molecule>
```

Similar interfaces are provided for all portions of CML commonly used by simulation codes.

Such calls do not interfere with the flow of the code, and it is immediately obvious what their purpose is. Further, with such calls, a developer with only a rudimentary knowledge of XML/CML can easily add CML output to their code.

This *wcml* interface is now used in most of the Fortran simulation codes within *e*Minerals, including the current public releases of SIESTA 2.0 and DLPOLY 3, and in addition is in the version of CASTEP[15] being developed by MaterialsGrid[16]. FoX is freely available, and BSD licensed to allow its inclusion into any products - we encourage its use.

4. Data visualization

Although reformatting simulation output in CML allows for much richer handling of the data, it has the aforementioned disadvantage that the raw output is then much less readable to the human eye than raw textual output.

This led to the desire for an tool which would translate the CML into something more comprehensible. At its most basic level, this could simply strip away much of the angle-bracketed-verbiage associated with XML. However, since a translation needs to be done anyway, it is then not much more trouble to ensure that the translated output is visually much richer.

Such tools have been built before on non-XML bases, with adapters for the output of different codes, but what we hoped to do here was, through the use of XML, avoid the necessity for the viewing tool to be adapted for new codes. In addition, it was strongly desired that the viewing, as far as possible, require no extra software installation from the user, in order that data could be viewed by colleagues and collaborators as easily as possible.

Since CML is an XML language, and translations between XML languages are easily done; and XHTML is an XML language; and furthermore, these days the web-browser is an application platform in its own right, which is present on everyone's desktop

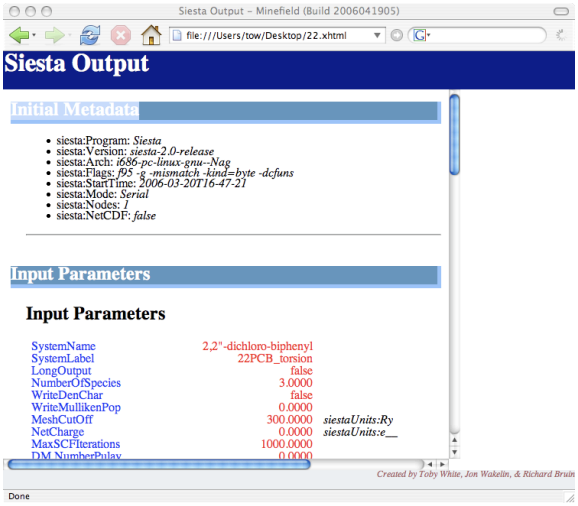

Figure 1: marked up metadata and parameters

already; we followed the route of transforming our CML into XHTML.

The browser application platform offers rich presentation of textual data, and rich presentation of graphical data, which can also be in an XML format, such as SVG (Scalable Vector Graphics). Furthermore, there is the possibility of interactivity, through the use of Javascript (JS) to manipulate the displayed XHTML/SVG. Finally, it affords the opportunity to to embed Java applets, for interaction on a level beyond what JS+XML can do; especially in this case through the use of Jmol[17], which is a pre-existing, well-featured molecular visualization platform.

Therefore, we wished to transform our CML into XHTML. This could be accomplished by many methods, but, XSLT was the obvious choice, since it is explicitly designed to convert between XML languages. Further, modern web-browsers have limited, but increasing, support for performing XSLT transformations themselves. This held out the possibility that we could rely on the browser to perform the transformation as well as the rendering of the output. We would then be able simply to point the web browser at the raw CML, and conversion would occur automatically. and in addition, is interpretable by the browser itself; thus it should be possible to view the CML file directly in the browser and have the transformation take place as the document is rendered.

The result of this process was a set of XSLT transforms which we term **ccViz** (computational chemistry Visualizer). Browser XSLT capabilities are sadly not yet at the stage where they can perform the XSLT, but nevertheless **ccViz** has proved invaluable.

The XSLT transformation starts with simple mark-up of quantities with their names and units extracted and placed together. Thus instead of looking through the CML to find the total energy, the name "Total Energy" is marked up in colour, and its value shown, with units attached. This is shown in figure 1. Although the resultant page is nicer to look at, and immediately more readable, this transformation is very straightforward. However, two particular further aspects of the transform deserve attention.

4.1 SVG graphs

Firstly, of note is the production of SVG graphs from the CML data. For a simulation output, it is valuable to see the variation of some quantities as the simulation progresses; of temperature, or total energy, for example. This can be done with a table of numbers, and indeed traditionally has been - the simulation scientist grows used to casting their eye down a list of numbers to gauge variation when viewing text output files, in the absence of a better solution. However, a line graph is much easier to grasp. SVG is ideally suited for rendering such graphs, as a 2D vector language. A generalized line-graph drawing XSLT library was written, which, when fed a series of numbers, will calculate offsets & draw a well-proportioned graph, with appropriate labels and units. The visualization transform can then pull out any relevant graphable data (which may be determined solely from the structure of the document, independent of the simulation code used), and produces graphs, which are then embedded inline into the XHTML document.

The transform is performed entirely within XSLT, without recourse to another language, which makes it embeddable in the browser engine. An example is shown in figure 2. The plotting engine is known as Pelote and is freely available for use.

Thus, a mixed-namespace XHTML/SVG document is produced, and on viewing the output file, it is immediately easy to see the progress of the simulation by eye. This is of great importance, augmenting productivity by increasing the ease with which the researcher may monitor the progress of their simulations, particularly in a grid-enabled environment, with many concurrent jobs running.

4.2 Jmol viewing

Since the outputs of all of our codes concern molecular configurations, it would be extremely useful to be able to see and manipulate the 3D molecular structures generated by the simulation. This is a task it is impossible to perform by eye from a listing of coordinates. However, there is no 3D XML language which is sufficiently widely

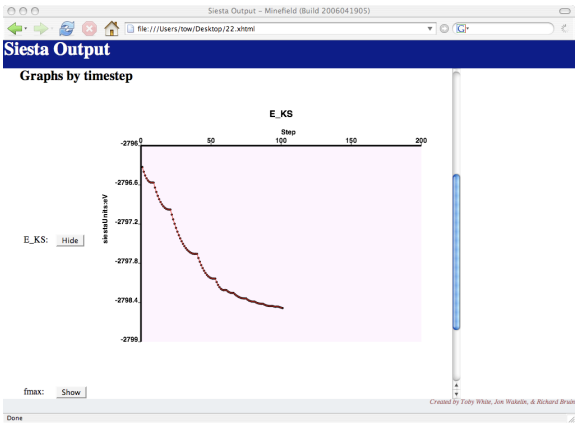

Figure 2: automatically generated SVG graph

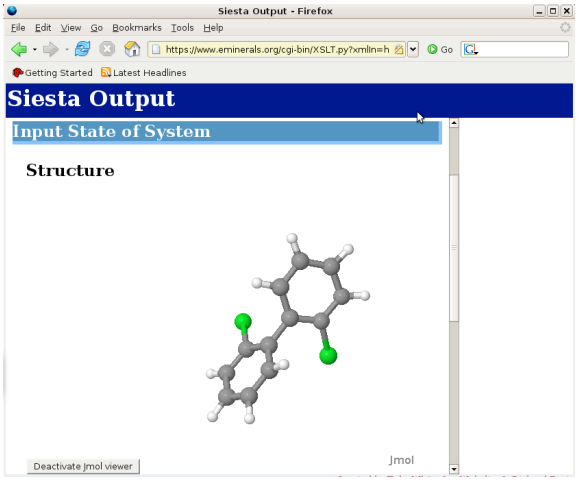

Figure 3: Interactive Jmol applet embedded in webpage.

implemented in browsers to make this doable in the fashion we managed for 2D graphs.

Fortunately, the well-established tool, Jmol[17], which knows how to read CML files, is primarily designed to act as an applet embedded in web-pages. We may therefore use our XSLT transform to embed instances of Jmol into the viewable output. However, Jmol accepts input only as files or strings, so we needed to find a way to pass individual different molecular configurations from our monolithic CML document to different individual Jmol applets embedded in the transformed webpage. We overcame this problem by producing multiple-namespace XML documents: the output contains not only XHTML and SVG, but also chunks of CML, containing the different configurations of interest. Since the browser is unaware of CML, it makes no attempt to render this data. We then took advantage of Java/Javascript interaction to enable its use.

There is an API for controlling the browser's JS DOM from a Java applet, called LiveConnect. We adapted the Jmol source code so that when passed a string which is the id of a CML tag in the embedding document, Jmol will traverse the in-memory browser DOM to find the relevant node, and extract data from the CML contained therein. This enables us to have a single XHTML document directly containing all the CML data, which knows how to interpret itself. This addition to Jmol has been included in the main release, and is available from version 10.2. An example is illustrated in figure 3.

4.3 Final document

Thus, we constructed an XSLT transform which, when given a CML document, will output an XHTML/SVG/CML document which is viewable in a web-browser; in which all relevant input & output data is marked-up and displayed; all time-varying data is graphed, and all molecular configurations can be viewed in three dimensions and manipulated by the user.

Very conveniently, the XSLT transform knows nothing about what these quantities are, nor from what code they originate. So, we may add new quantities to the simulation output, and they are automatically picked up and displayed in the browser, and graphed if appropriate.

Furthermore, it is important to note that any other code which produces CML output is viewable in the same way. This is useful because one of the biggest barriers to be overcome in sharing data between scientists from different backgrounds is a lack of familiarity with each other's toolsets. However, with ccViz, we can share data between DL_POLY and SIESTA users and view them in exactly the same way.

This transferability applies not only to this visualization tool, but to any analysis tools built on CML output - a tool which analyses molecular configurations, or which performs statistical analyses on timestep-averaged data, for example, need be written only once, since the raw information it needs may be extracted from the output file the same way for every CML-enabled code. Previously, such a tool would rely on the output format of one particular simulation code.

5. Data management

As mentioned above, working within a grid-enabled environment, it is easy to generate large quantities of data. The problem then arises of how to manage this data. Within the *e*Minerals project we store our data using the San Diego Supercomputing Centre's SRB [18], though there are many other solutions to achieve similar aims of distributed data storage.

However, the major problem encountered is not where to store the data, but how to index and retrieve it; for both the originator of the data and their collaborators. The *e*Minerals project faces a particular challenge in this regard, since we have a remarkably wide variety of data being generated. XML helps in this task in two ways. Firstly, having a common approach to output formats gives the advantages explained in the previous section. The second, larger, issue is the general problem of metadata handling, and approaches have been attempted with varying degrees of success by many people. The *e*Minerals approach is explained in great detail in [19]. However, we shall discuss it briefly here, with particular reference to the ways in which XML has helped us solve the problem.

There are three sorts of metadata which we have found it useful to associate with each simulation run:

- metadata associated with the compiled simulation code - its name, its version, any compiled-in options, *etc*. Typically there will be 10 or so such items.
- metadata associated with the set up of the simulation; input parameters, in other words. These will vary according to the code - some codes may use the temperature and pressure of the model system, some may record what methods were used to run the model, *etc.* Typically there will be 50 or so such parameters
- Job-specific metadata. Since the purpose of metadata is in order to index the job for later, easy, retrieval, sometimes it is appropriate to attach extracted output as metadata. For example, if a

number of jobs are being run, with the final model system energy being of particular interest, it is useful to attach the value of this quantity as metadata to the stored files in order that the jobs may be later examined, indexed by this value.

In each of these cases, we find our life made much easier due to the fact that our files are in an XML format.

For the first two types of metadata, we use the fact that CML offers **<metadata>** and **<parameter>** elements, which have a (simplified) structure like the following:

```
<metadataList>
  <metadata name="Program"
content="DL_POLY">
  ...
</metadataList>
```

We may therefore extract all such elements and store them as metadata values in our metadata database.

The third type of metadata obviously requires specific instructions for the job at hand. However, because the output files are in XML format, we may easily point to locations within the file such that tools can automatically extract relevant data. This can be done easily using an XPath expression to point at the final energy, for example (although in fact we do not use XPath directly - we use AgentX[20], as detailed in the next section.)

6. Data transfer

Finally, we have also succeeded in using XML to work towards code interoperability in the fashion originally foreseen.

The idea of a common data format is attractive, and as explained in section 2, would be of enormous value. It has, however, proved elusive so far, for a number of reasons, the discussion of which is beyond the scope of this paper. Nevertheless, by passing around small chunks of well-formatted, well-specified data, and agreeing on a common dialect with a very small vocabulary, we are able to gain much in interoperability.

Much of the progress we have made in this area is due to AgentX, an XML abstraction tool we have built to help us in this task.

6.1 AgentX

AgentX is a tool originating primarily from CCLRC Daresbury, but in the development of which *e*Minerals has played an important part. A previous version was described in [20].

At its most basic, it may be understood as, firstly, a method of abstracting complicated XPath expressions, and secondly as a method of abstracting access to common data which may expressed differing representations by various XML formats.

For example, the three-dimensional location of an atom within a molecule is expressed in CML, as shown in section 3.2 above, with nested **<molecule>**, **<atomArray>**, and **<atom>** elements. AgentX provides an interface by which one can access that data in terms of the data represented rather than the details of that representation, as illustrated by the following, simplified, series of pseudocode API calls.

```
axSelect("Molecule")
numAtoms = axSelect("Atom")
for i in range(numAtoms):
   axSelect('xCoordinate')
   x = axValue()
   axSelect('yCoordinate')
   y = axValue()
   axSelect('zCoordinate')
   z = axValue()
```

Internally, this is implemented by AgentX having access to three documents - the CML source, an OWL[21] ontology, and an RDF[22] mapping files.

"Molecule" is a concept defined in the OWL ontology; and the RDF provides a mapping between that concept and an XPath/Xpointer expression, which evaluates to the location of a **<molecule>** tag. Furthermore, the ontology indicates that "Molecule"s may contain "Atom"s, which may have properties "xCoordinate", "yCoordinate", and "zCoordinate". The RDF provides mappings between each of these concepts, and XPath expressions which point to locations in the CML.

Thus, presented with a CML file containing a **<molecule>**, it may be queried with AgentX to retrieve the atomic coordinates by a simple series of API calls, without the need to understand the syntax of the CML file.

More powerfully, though, it is possible to specify multiple mappings for one concept. That is, we may say that a "Molecule" may be found in multiple potential locations.

The normal CML format for specifying a molecule is quite verbose, albeit clear. For DL_POLY we needed to represent tens of thousands of atoms efficiently, so we used CML's **<matrix>** element to provide a compressed format, at the cost of losing the contextual information provided by the CML itself.

However, by simply providing a new set of mappings to AgentX, we could inform it that "Molecule"s could be found in this new location in a CML file, so all AgentX-enabled tools could immediately retrieve molecular and atomic data without any need for knowledge of the underlying format change.

AgentX is implemented as an application on top of libxml2. It is implemented primarily in C, with wrappers to provide APIs for Perl, Python and Fortran.

Concepts and mappings are provided for most of the data that are in common use throughout the project, but it is easy to add private concepts or further mappings where the existing ones are insufficient.

6.2 Data interchange between codes

We have incorporated AgentX into the CML-aware version of DL_POLY. This has enabled us to use CML-encoded atomic configurations as a primary storage format for grid-enabled studies. Details of studies performed with this CML-aware DL_POLY are in [23].

Furthermore, the output of any of our CML-emitting codes may now be directly used as input to DL_POLY, so we can perform direct comparisons of SIESTA and DL_POLY results.

AgentX has also been linked into a number of other codes., including AtomEye[24]; and the CCP1 GUI[25]. It is also used as the metadata extraction tool in the scheme described in section 5.

Thus we are now successfully using CML as a real data interchange format between existing codes that have been adapted to work within an XML environment.

7. Summary

Within the *e*Minerals project, we have made wide, and increasing, use of XML technologies, especially with CML.

While working towards the goal of data interoperability between environmentally-relevant simulation codes, we have found several additional areas where XML has been of particular use, and have developed a number of tools which leverage the power of XML technologies to enable better collaborative research and eScience. These include:

- FoX, a pure Fortran library for general XML, and particular CML, handling.
- Pelote, an XSLT library for generating SVG graphs from
- ccViz, an XHTML-based CML viewing tool.
- AgentX, an XML data abstraction layer.

All of our tools are liberally licensed, and are freely available from http://www.eminerals.org/tools

Finally, we have successfully developed methods for true interchange of XML data between simulation codes.

Acknowledgements

We are grateful for funding from NERC (grant reference numbers NER/T/S/2001/00855, NE/C515698/1 and NE/C515704/1).

References

[1] Murray-Rust, P and Rzepa, H. S., "*Chemical Markup Language and XML Part I. Basic principles*", J. Chem. Inf. Comp. Sci., **39**, 928 (1999);
Murray-Rust, P. and Rzepa, H.S., "*Chemical Markup, XML and the World Wide Web. Part II: Information Objects and the CMLDOM*", J. Chem. Inf. Comp. Sci., **41**, 1113 (2001).

[2] Garcia, A., Murray-Rust, P. and Wakelin, J. *"The use of CML in Computational Chemistry and Physics Programs"*, All Hands Meeting, Nottingham, 1111. (2004)

[3] White, T.O.H. *et al. "eScience methods for the combinatorial chemistry problem of adsorption of pollutant organic molecules on mineral surfaces"*, All Hands Meeting, Nottingham, 773 (2005)

[4] Soler, J. M. *et al*, " *The Siesta method for ab initio order-N materials simulation"*, J. Phys.: Condens. Matter, **14**, 2745 (2002).

[5] Todorov, I., and Smith, W., Phil. Trans. R. Soc. Lond. A, **362**, 1835. (2004)

[6] Warren. M.C. *et al., "Monte Carlo methods for the study of cation ordering in minerals."* , Mineralogical Magazine **65**, (2001); also http://www.esc.cam.ac.uk/ossia/

[7] M. G. Tucker, M. T. Dove, and D. A. Keen, J. Appl. Crystallogr. 34, 630 (2001)

[8] Watson, G.W. *et al., "Atomistic simulation of dislocations, surfaces and interfaces in MgO"* J. Chem. Soc. Faraday Trans., **92**(3), 433 (1996); also http://www.bath.ac.uk/~chsscp/group/programs/programs.html

[9] Bruin, R.P., *et al., "Job submission to grid computing environments"*, All Hands Meeting, Nottingham (2006), and references therein.

[10] This is not the only Fortran XML library in existence, - see also http://nn-online.org/code/xml/, http://sourceforge.net/projects/xml-fortran/, http://sourceforge.net/projects/libxml2f90 - but it is the most fully featured. Its output support surpasses others and it is certainly the only one with CML support.

[11] http://www.saxproject.org

[12] http://www.w3.org/DOM/

[13] Bray, T. *et al*., *"Namespaces in XML 1.1"*, W3C Recommendation, 4 February 2004

[14] Bray, T. *et al., "Extensible Markup Language (XML) 1.1"* ,W3C Recommendation, 4 February 2004

[15] Segall, M.D. *et al.,* J. Phys.: Cond. Matt. **14**(11) pp.2717-2743 (2002)

[16] http://www.materialsgrid.org

[17] http://jmol.sourceforge.net

[18] http://www.sdsc.edu/srb

[19] Tyer, R.P. *et al., "Automatic metadata capture and grid computing",* All Hands Meeting, Nottingham (2006) - in press.

[20] Couch, P.A. *et al*.,*"Towards Data Integration for Computational Chemistry"* All Hands Meeting, Nottingham 426 (2005)

[21] http://www.w3.org/TR/owl-features/

[22] http://www.w3.org/RDF/

[23]Dove, M.T. *et al., "Anatomy of a grid-enabled molecular simulation study: the compressibility of amorphous silica"*, All Hands Meeting, Nottingham (2006)

[24] Li, J., *"Atomeye: an efficient atomistic configuration viewer"*, Modelling Simul. Mater. Sci. Eng., **173** (2003)

[25] http://www.cse.clrc.ac.uk/qcg/ccp1gui/