

The XML-based eMinerals dataflow

| | |
|-------------------------------|--|
| Journal: | <i>Computing in Science and Engineering</i> |
| Manuscript ID: | CiSESI-2007-05-0051 |
| Manuscript Type: | Special Issue - Usable community grid infrastructures |
| Date Submitted by the Author: | 30-May-2007 |
| Complete List of Authors: | White, Toby; Cambridge University, Dept. Earth Sciences Artacho, Emilio; Cambridge University, Dept. Earth Sciences Bruin, Richard; Cambridge University, Dept. Earth Sciences Dove, Martin; Cambridge University, Dept. Earth Sciences Murray-Rust, Peter; Cambridge University, University Chemical Laboratory Todorov, Ilian; STFC, Computational Science and Engineering Department Tyer, Rik; STFC, eScience Wakelin, Jon; Bristol University, Advanced Computing Research Centre Walker, Andrew; Cambridge University, Dept. Earth Sciences Walkingshaw, Andrew; Cambridge University, University Chemical Laboratory |
| Keywords: | E.0.e, H.3.2.a, H.3.5.e, H.3.5.f, H.5.3.c, I.2.13, J.2.d, J.2.e |
| | |

The XML-based eMinerals dataflow

Toby O.H. White^{1,a}, Emilio Artacho^a, Richard P. Bruin^a, Martin T. Dove^a, Peter Murray-Rust^b, Iljan T. Todorov^c, Rik P. Tyer^c, Jon Wakelin^{a,2}, Andrew M. Walker^a, Andrew Walkingshaw^{a,b}, Dan J. Wilson^d

^a Department of Earth Sciences, Downing Street, Cambridge CB2 3EQ. United Kingdom

^b University Chemical Laboratory, Lensfield Road, Cambridge CB2 1EW. United Kingdom

^c Daresbury Laboratory, Daresbury, Warrington WA4 4AD. United Kingdom

^d Institut für Mineralogie, J.-W. Göthe-Universität, Frankfurt. Germany

Abstract

The eMinerals project produces large quantities of data from multiple atomistic simulation programs, and new methods have been required for handling this quantity of data. We have adopted the Chemical Markup Language (CML) as the basis of our information management strategy. In order to store, manage, visualize, and analyse these data, we have developed a series of XML-based tools, which this paper explains and details. *TobysSRB*, a usability-enhanced front end to our distributed data storage facility. *FoX*, a Fortran library which enables transparent XML/CML use from existing programs. *Pélote*, an XML language and implementation for encoding and representing 2D vector data, and *ccViz*, a browser-based CML visualizer.

1. Introduction

The full background to the eMinerals project has been explained in [REF]. The project is tasked with investigating a number of environmentally-relevant problems from the perspective of atomic-scale simulation, and in doing so, to take full advantage of eScience technologies, extending them as the need arises. This paper will discuss the response of the eMinerals project to its data-management problems, and the solutions that we have developed³.

Data management problems arose for three reasons.

- Firstly, the sheer quantity of data produced. Although our data production rates do not compare with projects such as the LCG⁴, nevertheless with the availability of usable grid middleware and job submission tools as described in the companion paper⁵, it rapidly outstripped the ability of scientists to cope with using traditional methods.
- Secondly, we faced the problem of a diversity of data. Unlike many projects in an otherwise similar situation, the sources of our data can be quite different in format and semantics. Although at some level they all relate to similar information (since eMinerals is studying a coherent field of science), they certainly cannot be trivially mapped onto a single predetermined data model.
- Thirdly and more subtly, eMinerals is a consciously cross-disciplinary project. We have physical scientists with backgrounds from different fields of chemistry, physics, and geophysics. This means

¹ Corresponding author. Email: tow21@cam.ac.uk, Tel: +44 1223 333464

² Now at: Advanced Computing Research Centre, Woodland Road, Bristol BS8 1UB. United Kingdom

³ This paper includes material previously presented at the 2nd IEEE International Conference on eScience and Grid Computing, Amsterdam, December 2006. Further information is available in the conference proceedings.

⁴ <http://lcg.web.cern.ch/LCG/>

⁵ "Job submission to grid computing environments", also in this volume.

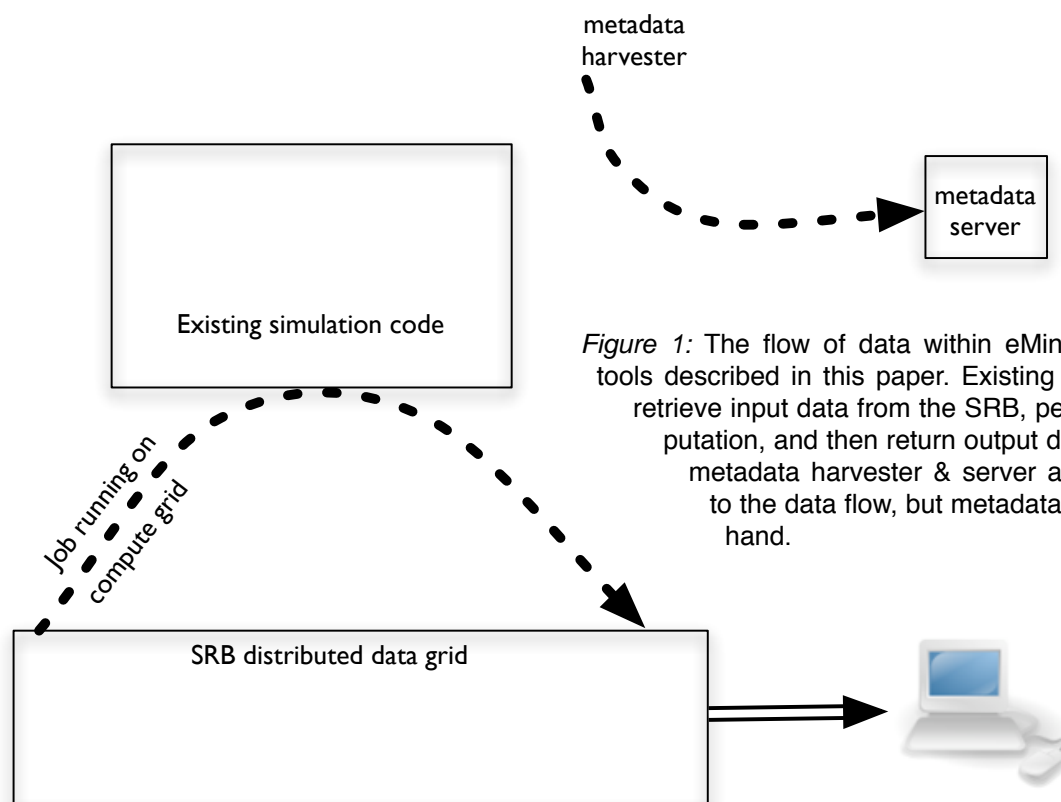


Figure 1: The flow of data within eMinerals without the tools described in this paper. Existing simulation codes retrieve input data from the SRB, perform some computation, and then return output data to the SRB. A metadata harvester & server are not connected to the data flow, but metadata can be added by hand.

that they are not all familiar with all of the data that is produced within the project - neither in terms of the formats involved, nor in terms of the variations in semantics between sub-disciplines.

The eMinerals project has a workflow involving job submission, data storage and metadata cataloguing, which is illustrated in Figure 1. However, this does not fully address the need for transfer of rich information (as opposed to streams of bits and bytes) - while it partly addresses the first point listed above, that of merely storing large quantities of data, more work was necessary to address the second and third points.

The project has developed a series of techniques and tools which overcome these obstacles. Firstly, the quantity of data, and the geographically-distributed collaborative nature of the project meant that some form of distributed data storage was necessary. Much of the background to this has been described in previous publications; here we talk in detail about the development of *TobysSRB*, a particular interface to our data storage.

Secondly, the bedrock of our data management strategy has been XML - in particular, the Chemical Markup Language (CML). The reasons for this are thoroughly explored in a companion paper⁶ - here it will suffice to say that XML offered us the flexibility, extensibility we required in a data format, while allowing us to leverage a wealth of pre-existing software and standards. However, in developing XML workflows around the previously non-XML universe of computational atomistic simulation, a significant quantity of additional software was needed.

We discuss firstly *FoX*, a library designed to bridge Fortran and XML. We consider separately two aspects of it. On the one hand, that devoted solely to exposing access to XML in a Fortran idiom, which we believe to be a contribution of general and wide value; and on the other, a carefully designed API entirely hiding the complexity of XML and CML from the user, allowing Fortran programmers to interact with CML documents despite having no knowledge of the format.

⁶ "Developing an XML ecosystem", also in this volume.

We also discuss some of the tools we have developed to consume our XML documents, allowing users to analyse and visualize CML-encoded data. In particular, we demonstrate *Pélote*, an XML language and associated toolkit for drawing two-dimensional graphs; and *ccViz*, a browser-based visualizer for CML documents.

In addition, though we do not describe it further here, we note that the XML-centric nature of the eMinerals data strategy has empowered our coupled job-submission and metadata system, as described in parallel papers⁷.

2. Distributed data storage

The eMinerals consortium has a widely geographically dispersed membership; and has a requirement for strong collaboration. These require that everyone have equal access to data. In addition, the computational minigrid will run across diverse resources, taking input from, and dumping output into, data servers distributed across the whole project. By using a distributed data grid, with a central access point, all project members are able to transparently access not only their own, but importantly, each others data. Without such a data policy, the effectiveness of the project would be drastically reduced.

Our data grid currently runs on the SDSC SRB⁸. The reasons for this, and our experiences with the SRB, have been described in a series of earlier papers⁹. The SRB is in wide use across the eScience community, but the eMinerals project was one of its earliest European adopters - our repository is now the largest in the UK, as measured by number of data objects stored.

By developing our project on the basis of the SRB, project members are now in the fortunate position that by default, none of our data is located on our individual computers; rather it is accessible in a network-visible filesystem. No longer are we tied to a single workstation - multi-homed working is much easier. More pragmatically, when working with a collaborator at their desk, it is easy to gain access to illustrative files, without cumbersome copying between computers. Equally, when a remote collaborator requests data, no longer do they need to wait while we to dig out files and email them, hoping any context remains clear. Rather, we can simply point our collaborator to a location on the filesystem, where the relevant files rest in context. Indeed, often a collaborator will not even need to request the data - it will be obvious and visible immediately it is produced.

However, we found that the SRB did not offer everything we needed. It is intended to offer a filesystem-like interface. It allows users to upload files and directories ("collections" in SRB parlance) so that they are visible to all SRB users. However, the SRB's 'filesystem-like' interface is not a true filesystem - that is, it cannot be manipulated using normal tools which understand native filesystems; rather interaction with the SRB must be performed through special SRB-aware tools which upload and download files to the local filesystem before viewing and manipulation. Furthermore, the provided SRB interfaces fell short in a number of ways. These shortcomings fell mostly into three categories:

- Poor UI design. This is most trivial in some ways, but in practice very important.
- The SRB is not well-suited to use across real-world networks, with restrictive firewalls in multiple institutions over which the project does not have full control. The SRB requires several non-standard open ports, and for every additional port that must be opened, we have to negotiate with computer officers in multiple departments in multiple institutions.
- The SRB interfaces (with one exception) all require installation on every machine from which the SRB is to be accessed - this strongly inhibits the ease-of-use that universally-network-accessible data ought to offer.

⁷ "Job submission to grid computing environments", and "Automatic metadata capture and grid computing".

⁸ <http://www.sdsc.edu/srb/>

⁹ Doherty, M., *et al.*, Proc. 2nd UK eScience All Hands Meeting, Nottingham, pp 51-58 (2003); Berrisford, P., *et al.*, Proc. 3rd UK eScience All Hands Meeting, Nottingham, pp 732-739 (2004)

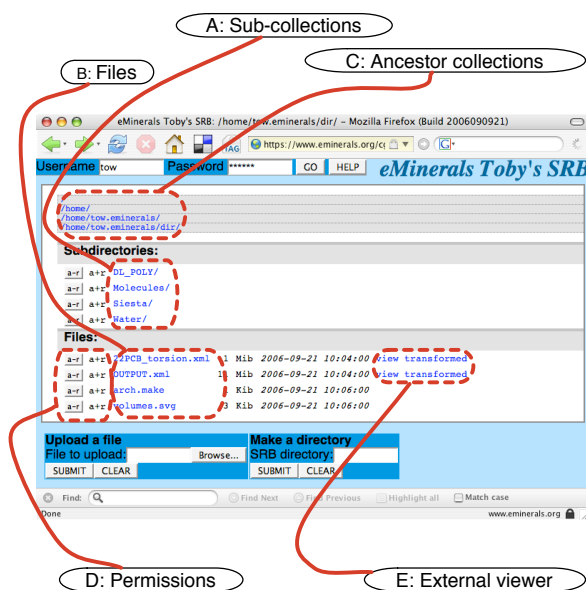


Figure 2: TobysSRB screenshot

requests onto the virtual HTTP filesystem into SRB requests from the web server to the SRB server. This means that on the server, all that is required is a CGI-enabled webserver; a version of Python (2.2 or greater) and, of course, access to the SRB. Thus, the firewall problem is reduced to ensuring a clear path from web server to SRB server, and client machines need only have ports 80/443 open to the web server.

On the client end, nothing is required except a tool which understands HTTP; a web browser if the user is human; if the user is a program, then *wget*, *curl*, or even raw socket access suffices.

An example of the visible web interface is shown in Figure 2. The interface looks as much like a graphical filesystem browser as can be built quickly & easily with HTML and CSS in a web browser. Files are represented as links - when clicked on, the file is served up, to be displayed in the browser or an appropriate helper application - or (since this is a standard web browser) right-clicking will allow saving the file to the desktop. Subdirectories are represented by links which, when followed, result in a similar display for the contents of the subdirectory. There are a number of other additional features aimed at increasing usability; these are described in full elsewhere¹¹.

In particular, though, note the additional link E in the diagram - such additional links appear whenever XML files are concerned, and are central to the eMinerals dataflow. We expand on them below.

3. Integrating Fortran and XML.

The computer language of choice of most computational minerals scientists is Fortran. Certainly, all of the simulation codes that are used within eMinerals are written in Fortran, and these codes are the source of all of our generated data.

As described above, we wanted to store all our data in an XML format - in CML in fact. However, we were presented with a number of Fortran codes producing data in various bespoke ASCII formats. If the simulation codes were written in one of several other languages, then the task of producing XML might have been fairly trivial, by means of plugging in an existing library to generate and serialize the appropriate XML. However, nothing of the sort existed for Fortran, which left us with a gap in

•The SRB is impossible to program against without installing client tools or libraries.

So we have built a new SRB interface which fulfilled our requirements better, called *TobysSRB*. It is web-based and operates through the browser. This addresses points two and three above, since only standard HTTP/S over ports 80/443 is required, and no installation is needed (since all computers come with a web browser). Point 1 was addressed by paying close attention to user requirements during design.

In addition, though, *TobysSRB* is carefully designed according to the principles of REST¹⁰, and essentially exposes the whole of an SRB filesystem as an HTTP filesystem, such that programs can address individual SRB data objects via HTTP requests.

TobysSRB is implemented as a simple CGI script, written in Python. The CGI script translates re-

¹⁰ "Representational State Transfer" - a philosophy for building lightweight web services. See R Fielding, "Architectural Styles and the Design of Network-based Software Architectures", PhD thesis, University of California Irvine (2000)

¹¹ White *et al.*, Proc. 5th UK eScience All Hands Meeting, Nottingham, pp. 209-216 (2006)

our data flow between the Fortran data-producing codes, and the rest of our XML workflow.

There are a number of approaches to bridging the gap between Fortran and XML, which have been taken under other circumstances. Where a very limited number of data sources are in use, and their output format is well-documented and static, then post-processing scripts can produce appropriate XML from ASCII output files. However, for a more diverse set of codes, with largely undocumented and varying outputs, this is less sensible, and it is not an extensible approach. Alternatively, Fortran wrappers can be constructed for existing XML libraries written in other languages. However, cross-language linking with Fortran is a business fraught with difficulty, especially when targeting multiple compiler combinations.

For these reasons, we have adopted a different approach. We have written our own, general-purpose XML I/O library in pure Fortran. With this approach the library can be simply linked in and called from any existing Fortran code with the minimum of further modifications. Of course, this does require having access to the source of the program - but with scientific codes this is almost universally the case. In any case, within eMinerals we are in the fortunate position for several of the major relevant codes. of not only having access to the source, but having active developers from each within the project team. We have thus been able to get our source modifications included in the released versions of these codes.

Our XML library is called *FoX* (for Fortran XML). An earlier version was described in [8]. It consists of several modules, governing various aspects of interaction with XML documents. Here we shall briefly describe only two of these, which allow the generation of XML.

The most widely-used part of the library is called WXML (because it Writes XML). This provides a number of API calls, which allow the direct construction of an XML document, and guarantees its well-formedness (optionally, it will also make certain guarantees about its validity, though full XML validity is not checked). Control over all aspects of XML document is offered, including DTDs and entities, with the unfortunate (but rarely scientifically important) exception of Unicode, which is not supported fully, since this is impossible within standard Fortran. WXML understands all of XML (and XML Namespaces) 1.0 and 1.1.

In so far as it is possible, WXML lets the user write directly to the XML Infoset, without concerning themselves about details of character escaping, or the precise syntax required for well-formedness. All of this is automatically taken care of by the library, unless the user tries to force output which is not representable within well-formed XML (for example, by not matching close tags with appropriate open tags), in which case the error is flagged verbosely, and output ceases. All of the burden of ensuring that the resulting document is well-formed is therefore taken care of by the library.

However, WXML does still need a fair degree of XML knowledge on the behalf of the Fortran programmer to be used effectively, and a high degree of knowledge of the XML format in question. Both of these are potentially problematic with respect to the long-term development of such simulation codes. Our solution to this problem is explored in the next section.

In addition to those described here, FoX provides several other modules, including some which provide interfaces for XML input to Fortran.

Further information is available at:

<http://www.uszla.me.uk/FoX/>

where up to date releases of source code, examples and full documentation is available, as well as links to a mailing list and support.

```
<molecule>
  <atomArray>
    <atom elementType="H" x="0.0" y="0.0" z="0.0"/>
    <atom elementType="H" x="0.74" y="0.0" z="0.0"/>
  </atomArray>
</molecule>
```

Figure 3: CML representation of a molecule containing 2 atoms. To produce even this small fragment requires 12 WXML calls.

4. Generating CML

A much deeper narrative explanation of the way in which we used and developed CML is given in a companion paper in this volume⁶. However, here we focus on the concrete ways in which our tools fed into, and were the result of this development.

WXML allows the production of well-formed XML output from Fortran. This succeeds in bridging the technical gap between XML and Fortran, by providing mappings between Fortran and XML data structures and idioms. However, the use of WXML still requires a fairly in-depth knowledge of XML on the part of the Fortran programmer, and this is undesirable for several reasons.

Typical Fortran programmers by and large do not understand XML. Furthermore they are often not full-time programmers, such that acquiring the background knowledge necessary to make best use of XML technology is outside the scope and time available for their computational work. In addition to this, a lot of scientific Fortran programs are collaborative efforts. Even if one developer on the team were sufficiently well-educated in XML to make use of WXML, the others would not be.

Furthermore, XML documents are typically quite verbose; objects in a given XML language are often composed of several tens of XML primitives - see for example Figure 3. Outputting data by manipulating the XML Infoset item-by-item requires a lot of WXML calls, which obfuscate and clutter up a program if inserted into existing code in a straightforward manner.

As a result, giving the FoX API to a typical computational chemistry Fortran programmer will not result in high-quality XML documents being generated. It is unlikely that most Fortran developers will be able, unaided, to make good use of WXML to output their data

Even if a few are - or if (as we have on eMinerals) outside developers are prepared to do the work and improve a code to use WXML - then

- The ugliness of the resultant code (with hundreds of verbose WXML calls) may result in the main developers refusing to accept such changes into their codebase.
- Clusters of such WXML calls are highly fragile to other developers accidentally breaking them; either by rearranging them such that they no longer correspond to well-formed output (eg, by mismatched tags) or such that, though they are well-formed XML, they are no longer valid fragments of whatever XML language is in use.

What we needed was an API which mapped as neatly and flexible as possible between the Fortran structures typically used in computational chemistry, and the requisite XML structures. In short, we wanted to be allow typical computational chemistry programmers to output to CML without needing to know any CML.

Thus, a layer was built on top of WXML, called WCML - again, because it Writes CML. The WCML API requires no knowledge of XML or CML at all. It provides subroutines which are clearly named, and do what they say, in outputting a given object. These subroutines accept as flexible as possible a variety of input arguments. Thus, for example, to output the molecular coordinates shown in figure 3, there is a call:

```
call cmlAddMolecule
```

The minimum information needed by this subroutine is the coordinates and name of each atom. Typically, these will be held as 1) a size-3xN array of double precision numbers, and 2) a length-N array of 2-character strings for the element symbols; and thus it can easily be called as

```
call cmlAddMolecule(xyz_coords, elem_names)
```

However, frequently, the coordinates are held, instead, as three length-N arrays, one each for the x, y, and z coordinates. Therefore, the programmer can also do:

```
call cmlAddMolecule(x_coords, y_coords, z_coords, elem_names)
```

Similarly, the numbers may be held as single-precision floats - again the subroutine can be called unchanged. By providing overloaded interfaces like this, WCML remains agnostic on how data ought

to be represented in the program - a choice that the individual scientific programmer is better informed (by reasons of efficiency or simple code-development history) than WCML. We reduce the burden on the programmer, who need not convert from their internal data representations to any preferred representation. Furthermore, such a process would be a potential source of error (both in the conversion, and in the duplication of information) Similarly overloaded interfaces are provided for all of the data objects which form the CML language.

Most importantly, in all cases, the programmer need not worry about precisely how to convert their coordinates into XML elements or attributes; there are no worries about misspelt names or badly ordered elements - WCML guarantees to take care of it and produce conforming CML output.

In fact, WCML's goal is that the developer need not care what CML is at all. It can be produced as above without any knowledge of the CML format, and then tools exist (and are described below) that process the CML appropriately. The user only needs to know that by issuing an appropriate `cmlAddSomething` call, then data is produced appropriately for correct manipulation thereafter.

This solves the problems mentioned above. In addition, this has the striking advantage that it brings XML output within the reach of even the casual developer. There is a culture, in scientific Fortran programming, of end-users making minor changes or additions to the code for their own purposes, without necessarily engaging the original developers, and without having any intention of making their work more widely available. Making the WCML API this accessible means that such end-users can quickly and easily include their output into the CML document without any need to educate them in the details of CML.

This has been a very successful policy, and the FoX library, with WXML and WCML APIs, is now included in the latest versions, or current development pre-releases of five of the most popular computational chemistry codes; SIESTA¹², DL_POLY¹³, GULP¹⁴, CASTEP¹⁵, and MOPAC¹⁶. Between them, these four codes have over 10,000 registered users. All of these users are not yet using the CML-enabled version of the code; certainly the vast majority of them will have no particular interest in the CML output format. However, it is our hope that since they will be creating CML output with no additional effort; indeed in some cases, without realising it, (SIESTA, for example, produces CML output by default unless it is switched off by the user), then a large corpus of CML documents will be built up without further effort.

In addition, work is ongoing in incorporating FoX into several other high-profile codes. An important extra measure of success though is that FoX has successfully been used to implement CML output for several mostly-locally produced codes; codes which have only a few users and no full-time developers. It is a mark of its ease-of-use that this has been feasible, proving that quick and easy XML output can be brought within reach of the casual scientific programmer.

5. Processing XML output

5.1 Data Analysis

Data analysis is largely a domain-specific problem, so we will not describe any of our tools here. Many of them are essentially bespoke scripts. However, their being built on an XML foundation enormously increased our power.

¹² <http://www.uam.es/siesta/>

¹³ http://www.cse.clrc.ac.uk/ccg/software/DL_POLY/

¹⁴ <http://www.ivec.org/GULP/>

¹⁵ <http://www.tcm.phy.cam.ac.uk/castep/>

¹⁶ <http://openmopac.net/>

Writing a script to analyse some data involves two separate jobs of work. Firstly, of course, the analysis itself must be encoded, whether this be as simple as averaging a quantity over a number of timesteps, or some complex statistical correlation. Before that can be done, though, the numbers on which the analysis is to be performed must be extracted from the data file. Here, XML helped us in two ways.

Firstly, the data extraction portion of the script becomes essentially identical for any CML output file, regardless of its origin. This means that we can immediately compare analyses between the outputs of different codes, rather than having to adapt analysis scripts for every code whose output we are interested in. Of course, this dramatically lowers the bar to open-ended data exploration - many more such analyses can be easily tried out in an experimental fashion.

Secondly, the data extraction portion of the script becomes much easier to write. Rather than a complex sequence of regular expression matching; combinations of *sed/awk/sh/perl*, etc., numbers can quickly and easily be extracted through an XPath API. Again, this makes such scripts easier to write - which means that scripts will be written where they weren't before.

One item of initial concern here was the accessibility of the XML output to non-XML scientists. It is important to remember that many data analysis scripts are written in a casual fashion, by scientists who need to quickly understand one particular set of correlations between data - no-one else may have been interested in it, nor may anyone else in the future. Thus there is a widespread practice of writing quick-and-dirty scripts for nobody's consumption but your own. We were at pains not to remove this power from the hands of the end-user in our zeal to promote XML.

This fear has proved unfounded. For one thing, while we have added CML output to codes, we have not removed the old, textual, output with which users are already familiar, so their job has been made no more difficult. Secondly, though, we have found that the idiom of XPath data extraction is easily grasped by scientist end-users.

We wrote WXML to insulate developers from the need to construct well-formed XML *output*, because there are a number of issues, some less than obvious, which must be paid attention to in order to guarantee XML well-formedness (without which the exercise is pointless). However, the situation is far less fragile where XML *input* is concerned. If users are presented with a valid, well-formed XML document, then there is much less education required for them to make use of it.

This has been borne out both informally, through collaborations between local and remote colleagues, but also in the formal classroom setting of a workshop which we recently ran¹⁷.

5.2 Data visualization

In contrast to the data analysis software we have written, some of our data visualization software we consider to be of sufficiently wide interest to be worth describing in detail. Here we shall consider two particular tools, which we call Pélote and ccViz.

5.2.1 Pélote

Very frequently in scientific data, there is a need to display vector arrays of data. By this we mean a series of (x,y) pairs, which mark the progression of some quantity, y , with respect to a second quantity, x . Often, x will represent time or space. There are many ways to display such data, but the simplest and most obvious is by means of a two-dimensional scatter plot.

However, there is no universal standard for encoding such information, nor for displaying it. Traditionally, such plots could be drawn in a graphing program, and exported to a raster graphics format. The raster output clearly loses all information about the original data, and is largely interpretable only by the human eye. Certainly later manipulation of the graph (by, for example, changing the value of one of the points, or rescaling the axes) is impossible with access only to the raster output. The graphing program may have some format which maintains the data, but in a largely inaccessible way.

¹⁷ "iFaX: Integrating Fortran and XML", National Institute of Environmental eScience, Cambridge, United Kingdom. 8th-10th January. 2007

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xml"
href="http://www.eminerals.org/XSLT/pelote.xsl"?>
<plot xmlns="http://www.uszla.me.uk/xsl/1.0/pelote/pelote.xsl">
  <pointList>
    <point x="-5.000000000000" y="24476.2352941176"/>
    <point x="-4.000000000000" y="24381.7843137255"/>
    <point x="-3.000000000000" y="24200.7647058824"/>
    <point x="-2.000000000000" y="24098.3725490196"/>
    <point x="-1.000000000000" y="23951.7254901961"/>
    <point x="0.000000000000" y="23835.0196078431"/>
    <point x="1.000000000000" y="23697.9803921569"/>
    <point x="2.000000000000" y="23569.9215686275"/>
    <point x="3.000000000000" y="23436.9607843137"/>
    <point x="4.000000000000" y="23267.2941176471"/>
  </pointList>
</plot>
```

Figure 4: An example Pélote file

Graphing programs may well also be able to output to a vector format, such as PS, PDF, or SVG - but while the structure of the plot is preserved in a more usefully machine-comprehensible and manipulable way, still most of the semantics are lost - any coordinates in the vector data are with respect to the page being drawn, not the original data.

We constructed a new XML mini-language (which we call Pélote) to represent such two-dimensional data in its own right, independently of what the data might mean; and furthermore, we wrote an implementation of an XSLT transform which converts this XML into a line-graph in SVG format. Because this implementation is performed entirely in XSLT, requiring nothing outside the world of XML, it is independent of any particular development and hosting environment.

By annotating a Pélote document with an `xml-stylesheet` PI¹⁸, we instruct any XML reader that the contents of the document should be visualized as a 2D plot, and if the XML reader is suitably equipped, the transformation can take place without any user intervention. Since the output is standard SVG, it can be viewed on any modern web-browser without the need to install further software.

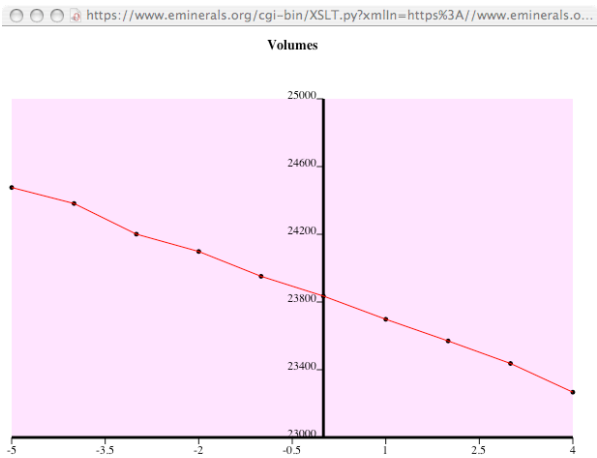


Figure 5: An SVG plot constructed by Pélote.

An example Pélote file is shown in figure 4, and its rendering in figure 5. It can be seen that the XML file contains no presentation data, simply a list of points in a very simple format which can be trivially created. The presentation (size, length, position of axes, and tickmarks, etc.) is inferred from the data by the pure XSLT transformation.

This format is very simple, and can be easily constructed by hand or by script. More complex output is possible if desired, for when the default presentation can be improved upon; the Pélote language contains directives to control axis/tickmark position, as well as line thickness, colouring, and so on. More importantly, since the original data lies unchanged in the source file, it can be manipulated directly.

¹⁸ <http://www.w3.org/TR/xml-stylesheet/>

5.2.2 ccViz

CML output allows for much richer handling of the data, and we have detailed several of its advantages in machine-processability above. However, it is undeniably harder on the eye than whatever textual data might previously have been the program's output, regardless of any improved retention of semantics in the data.

This led to the desire for a tool which would translate the CML into something more comprehensible, marking the output data up so that it can be better visualized. Such tools have been built before, on non-XML bases, to illustrate code output using whatever textual data is produced by a given code. However, such a tool will either be code-specific, or will require adapters for every code whose output it understands. Furthermore, it will suffer because the textual data it is using as input rarely is fully semantically marked up - the adapter for each code not only needs to adjust to different data formats, but needs to understand the implicit semantics of each code's output. We have written this tool, which we call ccViz (for Computational Chemistry Visualizer).

We wanted to write a viewer that could take full advantage of whatever semantics were available in the CML output, and in addition, would have no code-specific dependencies, so the same viewer could be used for any CML-formatted output. In addition, we strongly desired that the viewer, as far as possible, required no additional software installation from the user, and that it be platform-independent. This last was particularly important, since we wanted users to be able to share their data as easily as possible with collaborators and co-workers.

The easiest way to accomplish this was to leverage a web-browser. Nearly all modern web-browsers can be used essentially as XML viewing tools, which can render a number of XML languages by default (XHTML, SVG, MathML), and which can have viewers embedded (by, for example, Java, Flash, or Javascript) for additional XML languages. In addition, for all practical purposes everyone has such a web-browser on their desktop. This makes it an ideal host for the viewing environment we wished to build.

So we built our viewer by constructing a transformation from our CML output into a mixed-namespace XHTML/SVG/CML document. This is strictly an XML-XML transformation, so can be accomplished by XSLT, and again, the xml-style-sheet PI can be used to annotate our CML documents and inform XML applications that our CML documents can be rendered in this way.

From the perspective of our viewing tool, there are three sorts of data in our CML documents. Firstly, there is textual, or scalar numerical data. This is one-dimensional; it can be displayed simply by marking up the data in colour, italic, bold etc., displaying any units or other context as appropriate. This can all be accomplished with standard XHTML, as shown in figure X.

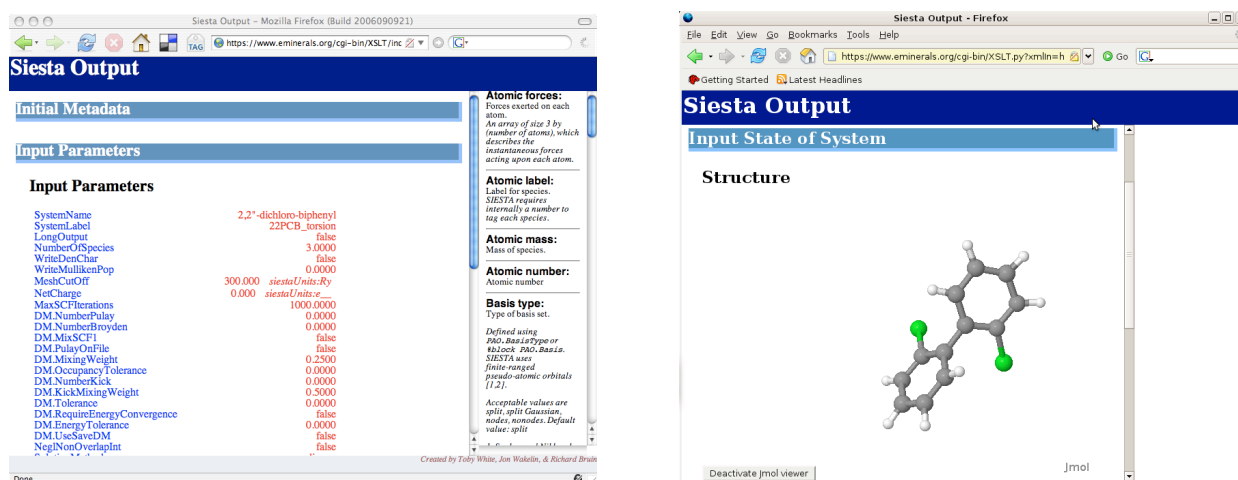


Figure 6: Examples of ccViz output automatically constructed from CML documents. On the left, marked-up textual output, with accompanying dictionary; on the right, an embedded applet showing 3D chemical structure, which can be interactively manipulated.

Secondly, there is two-dimensional data, which might be displayed as a table (using straightforward XHTML) or more usefully, as a line graph or histogram. The human eye can generally make more immediate sense of these latter than of a table of numbers. For some data, where tables are appropriate, they are displayed; elsewhere, Pélote was used to produce SVG graphs which were then embedded inline into the XHTML document.

Thirdly, there are molecular structures, which are three-dimensional entities. Although these could be printed out textually as a list of coordinates, they cannot sensibly be visualized in this way at all - they must be graphically and interactively displayed to be useful. Fortunately, there exists a tool, Jmol[REF] which will directly read CML files, and display molecular structures interactively; and Jmol is written so that it may be used as an applet, embedded in web-pages.

However, Jmol did not completely fulfil our needs; in its original form, it could not understand multiple namespace documents, nor could it read data from a document in which it was embedded. However, by using the LiveConnect API, we persuaded Jmol to pull its data from the browser-constructed DOM tree, which enabled us to embed multiple Jmol instances within our mixed-namespace XHTML document, allowing the end-user to visualize any CML fragments therein.

So our XSLT transform extracted any CML fragments representing molecular structures, and placed them directly inline to the XHTML output, with an embedded Jmol applet for each fragment. The result of visualizing such a document is shown in figure 6.

The end result was that our CML documents were annotated to direct that they could be rendered with this XSLT transform, so any XML readers can apply the transform directly, without human intervention. Alternatively, the transform can be performed, and the resulting XHTML document sent around between collaborators, who need know nothing of the underlying technology, and the output viewed, requiring no additional investment in time or software installation.

An important, and not entirely foreseen, outcome of writing ccViz is that there is a uniform platform for visualizing our CML output. The uniformity across codes now means that users who are familiar with one code now can usefully look at the output of another unfamiliar code. Without this common viewing paradigm, they would have had to go and learn the unfamiliar conventions of another output format, and all of its implicit semantics.

5.2.3 Integration into TobysSRB.

The advantage of using the xml-stylesheet PI is that in principle, any transform referred to can be automatically applied by an XML browser without end-user intervention. And indeed, most modern XML browsers include support for performing this XSLT transform. Unfortunately, however, as of writing this paper, this support is incomplete in all cases. Thus the transform must be run externally, which is a trivial one-line command, installed by default on all Linux and Mac OS X computers.

```
xsltproc output.xml > output.html
```

where the XML file may be Pélote, CML, or any other file annotated with an xml-stylesheet PI.

But users are not always happy with having to remember and use command-line tools, especially unfamiliar XML commands - and some end-users use Windows PCs, where it is not as simple. To this end we provide a user-accessible public web-page. This does nothing but provide a form to accept an XML document, apply a suitable transform, and present the result immediately back into the users browser. Users are easily educated that to view their output, they simply click on a bookmark and choose the file in the resulting dialogue box.

For users within our project, though, we have made the process even simpler. By default within the project, all output files are stored on the SRB. When users use TobysSRB to browse their SRB directories, then the directory listings are augmented where any XML files are found - an additional link is presented. As well as the normal link for downloading/viewing the file unchanged, a separate link is displayed which performs the transform on the server, and delivers the result straight to the user's browser. Thus the process is entirely transparent - from the users point of view, they simply ask to 'view as html' and the expected view is shown.

6. Conclusion

The eMinerals project has approached the problem of managing its data using a combination of a distributed data storage broker, and marking up its data in XML, specifically CML, which enables a host of complex informatics tools to be developed on this XML-based framework. Figure 7 illustrates

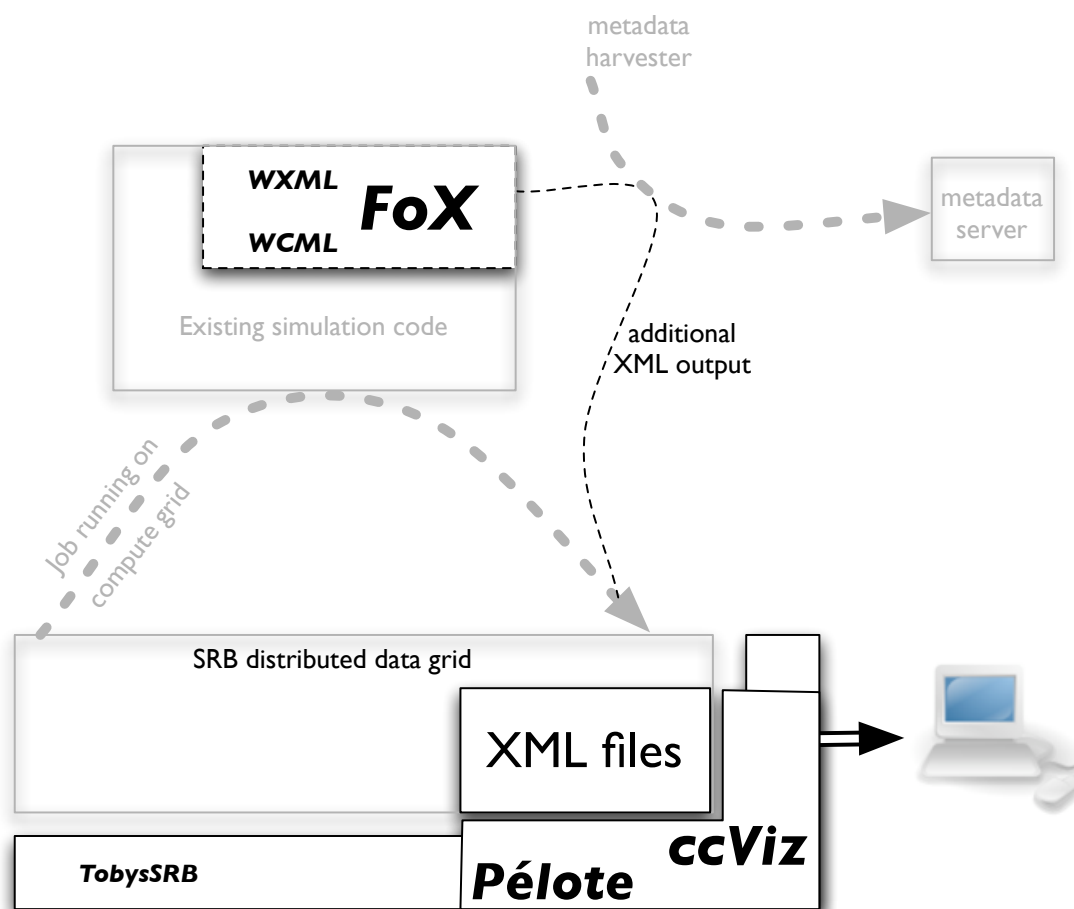


Figure 7: The flow of data within eMinerals with the tools described in this paper. Existing simulation codes retrieve input data from the SRB, perform some computation, and then return output data to the SRB. They also use *FoX* to output additional XML-encoded information. This can be picked up by the metadata harvester and added to the metadata database. Meanwhile, the XML file is stored on the SRB, from where it can be viewed through *TobysSRB*. Additional visualization is possible by transforming the XML using *ccViz* or *Pélote*.

how these tools are incorporated into the eMinerals system.

In this paper we discussed *TobysSRB*, a browser-based front-end to our data grid, which offers increased usability over previous interfaces, and requires nothing in the way of firewall configuration or client installation. It also allows the use of format-specific external viewers.

We also described *FoX*, a pure Fortran library for XML-enabling existing scientific programs. It has two components, *WXML*, which bridges the Fortran-XML gap, and *WCML*, which entirely abstracts away details of the XML allowing users to generate CML with the minimum of education.

Finally, we discussed the visualization of XML files, mentioning two tools - *Pélote*, a language and implementation for describing two-dimensional plots, and *ccViz*, which produces complex mixed-namespace XHTML documents to interactively visualize CML documents. Both of these are implemented in pure XSLT, and can be used as external viewers for *TobysSRB*.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

The end-user of the eMinerals dataflow can thus: use *FoX* to produce CML files from their simulation codes, which when run across the compute grid will deposit XML-encoded data in the distributed data repository. They can then use *TobysSRB* to explore the repository, and access XML files, which they can view through *Pélote* and *ccViz* for a rich interactive visualization of their data