



## XML AND COMPUTATIONAL SCIENCE

By George K. Thiruvathukal

**T**HE EXTENSIBLE MARKUP LANGUAGE (XML) IS A SPECIFICATION FOR DOCUMENT INTERCHANGE THAT THE WORLD WIDE

WEB CONSORTIUM (W3C) DEVELOPED IN 1998. IN

many ways, XML is the lingua franca among programming language enthusiasts, and proponents argue that it could potentially solve the multitude of data management and analysis problems the entire computing industry currently faces. These claims are not altogether new when it comes to computer science—remember when Java was going to end all platform discussions? But XML might make a real difference, especially in computing, engineering, and the mathematical sciences, in part because we can use it with different languages.

In this first article in a series about XML in computational science, I present some background and lightweight examples of XML usage, describe some XML component frameworks along with their purpose and applicability to computational science, and discuss some technical obstacles to overcome for the language to be taken seriously in computational science.

### XML Background and Examples

A good first step toward understanding XML is to take a brief detour into HTML, which everyone who did not hibernate during the dot-com boom of the 1990s knows and loves by now. In HTML, we compose Web pages by combining special markup tags—such as `H1` (header), `P` (paragraph), `PRE` (preformatted text), and so on—to provide content in a file (which usually has a `.html` extension). Web server software packages like Apache then serve this file, which is viewable in browsers such as Mozilla or Internet Explorer. Some of the markup tags, like `A` (anchor, which we use to make a hypertext link), have attributes that supply additional information. For example, `<A HREF="http://gkt-www.cs.luc.edu"> George's Home Page </A>` points to my Web site and displays a human-friendly link “George’s Home Page” in the browser.

HTML served its purpose during the dot-com era and will continue to do so, but several limitations became readily apparent almost as soon as it was released. In particular,

- the same framework addresses both content and appearance,
- browsers take liberties on how they interpret HTML (and therefore how HTML is rendered),
- the language itself provides no mechanisms for structuring information, so nonbrowser clients can’t easily process information in a structured manner.

Another key problem with HTML is that you can’t extract content from it. For example, you could automatically post a Web page with results from an experimental device. A human reading the page would see a result, say, “ $\pi = 3.14$ ,” but a computer program would find it daunting to identify that fact amid the HTML code.

All these HTML issues helped establish XML’s *raison d’être*. Essentially, we needed a framework that would allow a migration path from HTML; accordingly, the W3C declared that HTML would become compatible with XML. (XHTML is the W3C’s modular XML-based revision of the HTML standard, and most browsers already support it.)

The Hierarchical Data Format (HDF; <http://hdf.ncsa.uiuc.edu/>) provides a binary alternative to XML while managing to be compatible with several XML goals at the same time. In HDF files, the programmer can work with metadata to describe the binary data layout. Using various HDF API functions, the data can be traversed in a structured way.

Having said this, the decision to go this route perhaps makes most sense only at the model level (that is, in your scientific code proper). HDF availability is not likely to be ubiquitous on anything but Unix any time soon. Another alternative might be to use XML databases or embedding schemes that support on-the-fly compression. Many possible ways exist for implementing XML more concisely. The bloated appearance is just that—an appearance—and can have a different representation than a textual one. I’ll try to shed greater light on this in a future article.

In the rest of this section, we’ll go on a quick tour of XML’s core ideas and issues.

## Self-Describing Data

The XML community considers the notion of “self-describing data” to be the language’s cornerstone. But what does this mean? First and foremost, it means authors can encode data using any tags (called *elements* in XML speak) or attributes.

Suppose we’re writing code to do an *N*-body simulation, which is a well-known algorithm in computational astrophysics. In such applications, we commonly need a mechanism for specifying the initial positions in 3D space along with some parameters. Let’s look at how we could use XML here:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE NBody SYSTEM "nbody.dtd">
<NBody>
  <Parameters>
    <Parameter name="iterations"
      value="1000"/>
    <Parameter name="time-limit"
      value="3:00"/>
  </Parameters>
  <InitialConditions>
    <Particle id="mercury" mass="10" unit="g"
      x="10" y="100" z="25"
      velocity="100"/>
    <Particle id="venus" mass="10" unit="g"
      x="10" y="100" z="25"
      velocity="100"/>
    <Particle id="earth" mass="100" unit="g"
      x="100" y="200" z="30"
      velocity="20"/>
    <Particle id="mars" mass="100" unit="g"
      x="100" y="200" z="30"
      velocity="20"/>
    <Particle id="jupiter" mass="100"
      unit="kg"
      x="100" y="200" z="30"
      velocity="20"/>
    <Particle id="saturn" mass="10" unit="kg"
      x="100" y="200" z="30"
      velocity="20"/>
    <Particle id="uranus" mass="100"
      unit="kg"
      x="100" y="200" z="30"
      velocity="20"/>
    <Particle id="neptune" mass="100"
      unit="g"
      x="100" y="200" z="30"
      velocity="20"/>
  </InitialConditions>
</NBody>
```

```
<Particle id="pluto" mass="100" unit="g"
  x="100" y="200" z="30"
  velocity="20"/>
</InitialConditions>
</NBody>
```

As this example illustrates, one of XML’s strengths is its ability to describe ad hoc data and, subsequently, establish a formal structure behind it. Taking a slightly different perspective, think of being able to use variables in your code and assigning types to them later, thus letting you spend more time on the problem being solved and less time making the compiler happy.

The *N*-body example shows a typical data management need in scientific computing: embedding parameters and data in a file, which the application code interprets and processes. Such simulation data sets are usually encoded in a binary format, which makes it difficult for other researchers to run the simulation with their own data sets.

XML could bring more structure and flexibility to scientific data sets and processing. In our *N*-body data set, we configure the simulation application with a set of parameters and some initial conditions. In an *N*-body simulation, the initial conditions are a set of particles, each of which has a mass, (*x*, *y*, *z*)-coordinate in 3D space, and initial velocity.

One thing that is not yet apparent from looking at this XML is how you would incorporate it into your application. The answer, of course, is language-specific. A future article in this series will show how to integrate XML through parsing. For those who can’t wait, I have several materials and examples on my Web site (see also the “Coming Soon” sidebar).

Once we define an XML format for the application of interest, we quickly see that we could use the same format or adapt it to save a simulation’s results. For *N*-body-type simulations, which tend to run continuously, we might want to export the representation periodically to facilitate the simulation’s resumption at a later time.

## The Importance of a Literate Approach to Naming

XML’s proponents are among the first to suggest that one advantage is its ability to be viewed as plaintext. Although this might be true in theory, something viewable in plaintext is valuable only if it’s done in a literate and self-documenting style. The same is true with programming (because others might have to maintain your code), but the difference here is one of audience. Someone using your XML wants to run your *N*-body simulation, not think about the code behind it.

If XML is the intended interface, the content should be easy to create and modify. Otherwise, a tool should be de-

## Coming Soon

In the next article, I'll show you, in detail, how to recover data from XML files. If you have simple needs, you can use some standard tools. For example, this Python program prints each particle's mass and velocity in our `nbody.xml` file:

```
"""
particle_parser.py: A SAX parser to read our
particle data file and
create a data structure to hold the
particles (by name).
"""
import sys
import string
import types
import getopt

from xml.sax import saxexts
from xml.sax import saxlib

class ParticleHandler(saxlib.DocumentHandler):

    def startDocument(self):
        self.particles = {}
        self.parameters = {}
        self.context = []
        self.units = { 'g' : 1, 'kg' : 1000 }

    def startElement(self, name, attrs):
        self.context.append(name)
        elementPath = self.getContext()
        if elementPath == '/NBody/Parameters/Parameter':
            name = attrs['name']
            value = attrs['value']
            self.parameters[name] = value
        if elementPath == '/NBody/InitialConditions/Particle':
            id = attrs['id']
            unit = attrs['unit']
            mass = float(attrs['mass']) * self.units[unit]

            print id, unit, mass
            velocity = attrs['velocity']
            self.particles[id] = (mass, velocity)

    def endElement(self, name):
        self.context.pop()

    def getContext(self):
        return '/' + string.join(self.context,
"/")

    def printNicely(self):
        print """ Parameters """
        for (name,value) in self.parameters.items():
            print "%s=%s" % (name, value)
        print """ Particles """
        for (id, info) in self.particles.items():
            print "%s: mass %s velocity %s" % (id,
            info[0], info[1])

def go(particleFileName):
    parser = saxexts.XMLValParserFactory.make_parser()
    handler = ParticleHandler()
    parser.setDocumentHandler(handler)

    particleFile = open(particleFileName)
    try:
        parser.parseFile(particleFile)
    except IOError,e:
        print "Error", particleFileName, str(e)
    except saxlib.SAXException,e:
        print "Error", particleFileName, str(e)
    handler.printNicely()

if __name__ == '__main__':
    go(sys.argv[1])

This code demonstrates how to parse the XML and use the Document Type Definition (DTD) shown earlier. We will explore this in greater depth in an upcoming article on parsing.
```

signed to generate it. On the latter point, I encourage you to look at the MathML pages at the Wolfram and Associates site ([www.mathmlcentral.com](http://www.mathmlcentral.com)). MathML is a W3C standard that addresses both the presentation and content of mathematical formulae. The Wolfram site offers a Web service (or application) for generating MathML examples. You can get an immersive knowledge of MathML from this indispensable feature without having to read the specification, which amounts to several hundred pages of text.

XML elements are similar to objects in object-oriented programming languages, so you should use nouns to name

them whenever possible. The Smalltalk and Java communities tend to follow a naming convention that I also use in my examples. When a name has one or more words, each word is capitalized (Particle, InitialConditions, MyFavoriteElementName). The same rule applies for attributes, except for the first word, which is not capitalized; `xCoordinate` and `yCoordinate` are longer and more descriptive names than `x` and `y`. In short, if anyone other than you is going to use your XML, think about whether humans can parse it (without having to actually use an XML parser). All-lowercase-XML element names are very common, too. If you use lowercase

for attribute names, I suggest using C-style naming, where the words are separated with underscores.

## Schemas

Creating your own element names and attributes is a powerful concept, but it causes a major problem: How do you enforce the use of certain tags? How do you ensure that every `Particle` has the same (required) attributes? XML addresses these questions through *schema*.

Schema definitions are innate to XML. Every XML document can create one by embedding or referencing a Document Type Definition. DTDs bear many similarities to regular expressions and context-free grammars (while managing to be neither).

Let's look at a DTD that describes the structure of the `NBody` document from earlier:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ELEMENT NBody (Parameters,
    InitialConditions)>
<!ELEMENT Parameters (Parameter*)>
<!ELEMENT Parameter EMPTY>
<!ELEMENT Particle EMPTY>
<!ELEMENT InitialConditions (Particle,
    Particle+)>
<!ATTLIST Parameter name CDATA #REQUIRED-
    value CDATA #REQUIRED>
<!ATTLIST Particle id ID #REQUIRED
    mass CDATA #REQUIRED
    unit (g | kg) #REQUIRED
    x CDATA #REQUIRED
    y CDATA #REQUIRED
    z CDATA #REQUIRED
    velocity CDATA #REQUIRED>
```

Some syntax describes other content such as entities, but I won't cover it in this introductory article. For this discussion, we'll use the DTD syntax to define elements and attributes:

1. An `NBody` is a sequence (the comma) of one `Parameters` element followed by one `InitialConditions` element.
2. A `Parameters` element might contain zero or more (the closure or star operator from regular expressions) nested `Parameter` elements.
3. An `InitialConditions` element must contain at least one `Particle` followed by one or more `Particle` elements. This lets you enforce the use of two or more particles for the simulation. Enforcing three or more requires

an additional `Particle` at the beginning of the sequence.

4. A `Parameter` element has required name and value attributes. The name and value must be so-called CDATA (arbitrary character data). You cannot validate this parameter's actual content with the DTD. The application would need to check whether the parameter name and value pair is valid, which involves customizing the parsing behavior.

5. A `Particle` element has required *id*, *mass*, *unit*, *x*, *y*, *z*, and *velocity* attributes that can be assigned arbitrary character data values. Again, the application would need to validate the data—check that the *x*, *y*, and *z* values are valid floating-point literals. The attribute *id* is special because we can use it to uniquely identify a `Particle` within the document. When the parser reads the document, it will perform the uniqueness check and remove any duplicates. The `Particle` element also has a required unit attribute, which is a restricted form of CDATA known as a NOTATION. Here we restrict the input to "g" and "kg" for grams and kilograms, respectively.

The DTD lets us model structures with a clear-cut syntax, but we must consider some limitations. First, the syntax is intended to be straightforward, allowing primarily for structural checking. Attributes can be checked only minimally. For example, we can enforce that an attribute be limited to a set of values (true or false, using NOTATION instead of CDATA), but there is no way to enforce that an attribute be expressed in double precision. Strangely, there is a way to ensure an attribute has numeric characters (NMTOKENS instead of CDATA).

Second, the syntax is intended for single-namespace applications, which is inadequate at times. You might want to use element names common to another markup language (such as HTML) within the document, for example. Although scientific applications tend to work just fine with single namespaces, other applications might not. For instance, a geographic information system would have information about an image (from GPS) but might require metadata to describe its features (possibly in generic HTML). This would require a more advanced mechanism such as XML Schema, which is a powerful framework with much greater complexity than the DTD syntax.

## Supporting Irregular Data Structures

The *N-body* problem is an example of how to apply XML to a reasonably straightforward structure: an array of particles. XML can address regular problems such as this with ease, although some might argue (legitimately) that using

XML imposes a significant overhead. The storage of the element and attribute names, along with the metacharacters found in XML, can easily result in document bloat, which could be prohibitive in very large-scale applications.

XML can play a major role in computational science algorithms with irregular data structures, such as molecular modeling, game trees, sparse matrices, and so on. Because the data in XML are tree-structured, we can use it to model just about anything.

Consider the notion of a sparse matrix, which is a data structure containing mostly zero data. A common technique is to use a form of run-length encoding of the run's start position (row, col), the number of items in the run (one by default), and the data, separated by whitespace. The following would be an example of a sparse matrix encoding in XML (with the associated DTD):

```
<?xml version="1.0"?>
<!DOCTYPE SparseMatrix SYSTEM "sparse.dtd">
<SparseMatrix rows="10" cols="100">
  <Run r="1" c="3" len="2">35.0 71.5</Run>
  <Run r="2" c="5">100.1</Run>
  <Run r="2" c="98">75.0</Run>
  ...
</SparseMatrix>
```

```
sparse.dtd:
<!ELEMENT SparseMatrix (Run*)>
<!ATTLIST SparseMatrix rows CDATA
                    #REQUIRED
                    cols CDATA
                    #REQUIRED>
<!ELEMENT Run (#PCDATA)>
<!ATTLIST Run r CDATA #REQUIRED
              c CDATA #REQUIRED
              len CDATA #IMPLIED "1">
```

Here's a way to encode a BandedMatrix in XML (with the associated DTD):

```
<?xml version="1.0"?>
<!DOCTYPE BandedMatrix SYSTEM "banded.dtd">
<BandedMatrix rows="100" cols="100"
              band_width="3">
  <Row>100 25.8 37.5</Row>
  <Row>27.5 33.5 110.5</Row>
  ...
</BandedMatrix>
```

banded.dtd:

```
<!ELEMENT BandedMatrix (Row*)>
<!ELEMENT Row (#PCDATA)>
<!ATTLIST BandedMatrix rows CDATA #REQUIRED
                    cols CDATA #REQUIRED
                    band_width CDATA
                    #REQUIRED>
```

This is merely a glimpse of what's possible. The most notable aspect is that the data set is self-describing, so we can know how it's encoded. Furthermore, we can systematically transform one set into another as appropriate via a framework known as Extensible Stylesheet Language Transformations (XSLT) without forcing the user to write any extra code.

## XML Component Frameworks: A Preview

My goal with this introductory article is to focus on the basics and set the stage for a series of articles on XML's advanced topics. Here's a glimpse of what I'll discuss down the line.

### Namespaces

Doing anything advanced with XML requires the namespaces feature; we've assumed a single namespace throughout this article. The concept of multiple namespaces might seem simple for any scientific programmer familiar with C++'s namespace or Java's package keywords, but it involves more than what meets the eye. Among other things, you must learn a new and confusing concept known as the uniform resource identifier (URI) and take steps to ensure its uniqueness.

### DOM and SAX Parsing

Most serious uses of XML require you to develop a custom parser. In many ways, the term parser is poorly chosen, because code for the parser is already written for you. Implementations exist for C, C++, Python, Java, and C# (Fortran programmers can use the C parsers). The W3C provides two types: the document object model (DOM) and the simple API for XML (SAX). Both approaches are very low level but require some expertise to learn.

### Transformation

Clearly, we need a framework to let us rewrite XML documents in other (possibly non-XML) representations. Suppose you had data organized as a DiagonalMatrix and wanted to reorganize it as a SparseMatrix. (After all, a DiagonalMatrix is really a SparseMatrix at heart.) You can do it in one of two ways:

## Dave's Sideshow

### SO SUE ME

Lately, it seems like intellectual-property litigation has replaced much of the software industry's innovation. You don't have to go far to read all sorts of stories about patent lawsuits (such as the recent Eolas victory over Microsoft), trade-secret lawsuits (the case of SCO versus, well, everyone who's ever touched Linux), and the Recording Industry Association of America (which is suing everyone who's left). I must admit that this whole state of affairs is more than just a little demoralizing.

It's also somewhat sobering to think that almost everything I did in my early days of computing is now illegal. At one point, I was completely fascinated by the inner workings of computers and taught myself assembly language. Wanting to know more, I spent a considerable amount of time disassembling the Apple 2 disk operating system so I could figure out how it worked.

I also spent a fair amount of time reverse engineering video games—mostly so I could modify them in curious ways. I also once devised a patch to an operating system that tied the computer's apparent "speed" to the game controller (so a user could speed up or slow down programs as needed by turning a dial). Admittedly, this wasn't the most practical software enhancement I've ever created, but that wasn't really the point (although maybe such a feature could dynamically adjust a system's position on the top 500 list). I learned a lot about computers by messing around with them. All those skills proved to be useful when I started working on scientific applications



that used "real" computers such as the supercomputers at Los Alamos.

Fortunately, most of these activities are now criminal offenses. I mean, that kind of antisocial behavior just shouldn't be tolerated. Truth be told, of course, I was never all that deviant—when I wasn't spending my time figuring out how programs worked, I was writing and promoting shareware and freeware. (Hmmm...maybe I was deviant after all.)

Ironically, I recently got to apply the skills from my "notorious cybercriminal" past in the context of an intellectual-property lawsuit, of all things. To make a long story short, I was contacted out of the blue to take a quick look at a couple of large C++ programs. So, I spent some time ripping them apart and reported back, "So, let me get this straight. This moronic fight is about two enormous programs that solve a totally trivial problem, which could have been solved by just about any programmer using a couple of simple SQL database queries?" Needless to say, I'm not sure my analysis flattered either side. However, I did learn one thing: some people not only like to solve trivial problems using questionable tools, but

they like to write business plans and fight about them as well. Oh, and I learned that the good guys usually lose—even when they win. You really have to wonder about all those other lawsuits. Sigh.

#### A Farewell

I'm sorry to report that this is my last article with *CISE*. I've enjoyed working with Paul over the years in addition to collaborating with him on the Scientific Programming department. I'd just like to thank Paul and everyone else for their support—may all your programs be bug-free (and litigation-free)!

- Write your own parser, build a data structure, transform the data structure, and generate the new representation (complicated).
- Use XSLT to define a set of rewriting rules that can perform the transformation (complicated for non-CS people, but easier than the first approach).

Examples of how to use both techniques will appear later in this miniseries.

#### XML Schema Recommendations

XML Schema expands the DTD concept to address some deficiencies. Most importantly, it works with namespaces

and lets you do more precise attribute-value checking. For scientists to be able to share models and results, using XML Schema will be imperative, given that different DTDs cannot be combined easily. Unfortunately, the cost of using XML Schema is significantly greater complexity. (There is a generic term, XML schema languages, which includes both the W3C XML Schema recommendations and the DTD. Regrettably, the W3C XML working group did not come up with a better name. Worse, the issue could get more complicated in the future. Dissent is already brewing because many feel that the XML Schema recommendations are not the best way to describe XML structure in general.)

## Additional Web Resources

If you're out on the Web looking for additional resources, these sites should get you started in the right direction:

<http://gkt-www.cs.luc.edu> is my Web site, which contains numerous examples of how to use XML from Java and Python.

You can find this article's examples here as well.

[www.w3.org](http://www.w3.org) includes all the World Wide Web Consortium's standards work, including the definition of the XML standard and its component frameworks.

[www.python.org](http://www.python.org) mentions XML many times. Although you might be familiar with it from previous issues, I'm including it for completeness.

[www.sourceforge.net/projects/pyxml](http://www.sourceforge.net/projects/pyxml) leads you to PyXML, which you'll need if you want to do XML parsing and other advanced frameworks from Python.

[www.jamesclark.com](http://www.jamesclark.com) is a site from one of the first to develop a working parser for XML. Clark's parser, written in C, is used just about everywhere, including in the PyXML package.

[www.w3.org/Math/](http://www.w3.org/Math/) provides many examples and services for MathML. It also shows an example of best practices with XML.

### Standard Markup Languages

Various specialized markup languages have emerged from standardization processes in recent years. I alluded to MathML, which is likely to be of interest for computing in science and engineering. But several other languages, such as the Chemical Markup Language, financial markup languages, Scalable Vector Graphics (SVG), and many others, might be more appropriate for your work.

**X**ML is not without its share of issues and potential pitfalls, but we can overcome many of its problems through advanced tools and research.

The W3C originally designed XML to be a plaintext specification. However, with the emergence of Web services and the need for random access to content within an XML document, it might be time for the W3C to consider a binary and nonprintable representation for XML. (It is often desired to change a small part of an XML document. Today's XML requires rewriting steps to make such changes, which can be inefficient in larger documents.) Legitimate reasons abound as to why XML must always be plaintext-viewable at some level; however, this need seems to be most prominent when displaying results—for example, in a Web browser or print media—or in the debugging and development of XML-based services.

Compression is another major challenge facing the XML community, but approaches to dealing with it depend on the need. Using gzip can compress a document to nearly the size of an equivalent binary representation. However, when the document needs to be processed, it still must be uncompressed, resulting in slow read/write performance. Another approach is to use an embedded database or representation that maintains parts of the document in compressed form. This allows for faster read/write performance, but nowhere near optimum compression. Yet another approach is to take

advantage of file systems with native compression. This keeps the application oblivious to persisting storage and I/O performance problems.

Unfortunately, XML seems to be well on its way to repeating one of the Web's common problems: no standard directory or repository mechanism for finding useful DTDs and schemas. Another troubling aspect of the language is how many ways exist to say the same thing—sometimes badly. Clearly, expertise will evolve in different application domains to define and design XML schemas. Although there must be a standard way to publish and discover available DTDs and schemas for different problem domains, it's not clear that the W3C realizes the importance of doing so. We can only hope that a better standardized mechanism than a search engine will emerge for working with XML document data.

Finally, we come to data binding, which is a major problem when working with legacy languages such as C and Fortran. XML clearly presents a view of the world wherein document content is tree-structured. Although mapping XML document data to and from tree data structures in C and the latest version of Fortran (with structures) is clear-cut, working with tree structures is unnatural to most scientific programmers. One of my ongoing XML projects is aimed at addressing this particular topic. I provide a tool (SimpleDigest) that lets you process XML documents using simple path selection (similar to the directory concept in Unix) and mapping to your own data structures. (For more on this project, see <http://gkt-www.cs.luc.edu/research/xir.>)

**George K. Thiruvathukal** is a visiting associate professor at Loyola University Chicago. He is also president and CEO of Nimkathana Corporation, which does research and development in high-performance cluster computing, data mining, handheld/embedded software, and distributed systems. He wrote two books with Prentice Hall covering concurrent, parallel, distributed programming patterns and techniques in Java and Web programming in Python. Contact him at [gkt@cs.luc.edu](mailto:gkt@cs.luc.edu).