Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Using and Abusing XML

Diomidis Spinellis

Words are like leaves; and where they most abound,
Much fruit of sense beneath is rarely found. —*Alexander Pope*

I was recently gathering GPS coordinates and cell identification data, researching the algorithms hiding behind Google's "My Location" (www.google.com/gmm/mylocation.html) facility. While working on this task, I witnessed the great interoperability benefits we get from XML. With a simple 140-line script, I converted the data I gathered into a de facto standard, the XML-based GPS-exchange format called GPX. Then, using a GPS-format converter, I converted my data into Google Earth's XML data format. A few mouse clicks later, I had my journeys and associated cell tower switchovers beautifully superimposed on satellite pictures and maps.

Convenient versatility

XML is an extremely nifty format. Computers can easily parse XML data, yet humans can also understand it. For example, a week ago a UMLGraph user complained that pic2plot clipped elements from the scalable vector graphics (SVG—another XML-based format) file it generated. I was able to suggest a workaround that modified the picture's bounding box, which was clearly visible as two XML tag attributes at the top of the file.

Furthermore, a simple tool can trivially determine whether an XML document is well formed (meaning that it follows XML's rules). And, if we have the document's schema (a formal description of a specific document's allowed composition such as GPX), we can validate that a given file follows the schema. These properties are a boon to interoperability. With the XML schema at hand, when we stumble across a data transfer problem between two applications, we don't need to quarrel about whose program's fault it is. A third party, an XML validator utility, can judge whether the data follows the schema and impartially assign the fault to the data's producer or consumer.

XML also gives our code more robust input handling. Input processing is a notorious source of bugs, because there are literally infinite ways to provide wrong input to a program. Moreover, malicious adversaries deliberately craft input data aiming to crash a program, or worse, gain and exploit its privileges. By using XML, we can solve this problem by relying on the widely available libraries for parsing our input. These libraries are, by design and through their ubiquitous deployment, much more resistant to abuse than any special-purpose code we could concoct on our own.

Finally, by adopting XML, we can take advantage of the scores of tools that work on arbitrary XML documents. Common tasks—like editing, validation, transformations, and queries—become just a matter of selecting and applying the right tool. Also, we can then apply the experience we gain with these tools on other documents we come across in our work. And if, like me, you're a devoted user of the Unix toolchest, have a look at XMLgawk. It manages to combine gracefully exactly what its awkward name suggests.

Best practices ...

When we use XML, we sacrifice (sometimes significant) processing time and space to gain interoperability. So, it makes sense to actually verify that we've achieved our goal. Once you come up with a schema, ensure that you have at least one independently written program to read and write data in that schema. Additionally, have a human edit the file and verify that its structure is intuitive to someone unfamiliar with the schema and that programs can still read and process the edited file. Also, formally document your schema in a schema language, such as RELAX NG or XSD (XML Schema Definition), and then have a third-party tool validate your XML files.

Another way to promote interoperability is to adopt existing schemas. You can do that either wholesale, by having your application read and write its data in an already existing schema—for instance SVG—or piecemeal, by having parts of your XML document follow widely adopted standards. For example, the schema for GPX uses the XML Schema `xsd:dateTime` data type for time stamping waypoints. In turn, this data type is precisely defined by reference to ISO 8601, the international standard for date and time representations. This approach lets you reuse large swaths of existing work and avoids troublesome ambiguities. A criticism of the Office Open XML file format is that it doesn't use existing standards for many of the elements it represents, such as (you probably guessed it) dates but also drawings.

Furthermore, try to make your program's XML output accessible to non-XML tools and humans. Specifically, if your data consists of records up to, say, 80 characters long, fit each one on a single line. This lets many line-oriented tools like Unix's wc, awk, sed, and grep process your data. In more complex files, use appropriate indentation to make the file's structure apparent to its human viewers.

… and tar pits

By far, the worst offense in the take-up of XML is its adoption as a format for human-produced code. Three representative examples are the Apache Ant build files, the XML schema definitions (XSD), and the eXtensible Style Sheet Language Transformations (XSLT). XML is an adequate, if verbose, format for data that programs produce and consume but a nightmare for humans looking at anything more complex than what can fit on a screen. In most programming languages, tokens get a large part of their meaning from their context. For instance, a word appearing on the left of an open bracket is a function or method name. Contrast this with XML, where each token is explicitly assigned its meaning through tags and attributes. For example, in a make file, we can associate a value with a variable by writing

```
TESTSRC=test/src
```

Placement on one side or the other of the equals sign distinguishes the variable from its value. In the corresponding XML-based Ant build file, we write the equivalent as

```
<property name="testsrc" location="test/src"/>
```

In this case, named attributes specify what's assigned to what. This XML's approach simplifies the parsing of arbitrary files, but the corresponding verbosity hinders comprehension and comfortable programming.

In computer languages, there's a sweet spot between conciseness and wordiness. Apparently, it's the place where the means for expressing an idea matches our cognitive ability. Languages occupying this spot are the ones in which we achieve long-term productivity (this includes maintenance). Some languages or programming styles, like APL and Perl one-liners, have strayed to extreme conciseness. Other languages, like Cobol and XML, err toward excessive wordiness. Both extremes hinder the software's analyzability, changeability, and stability and, therefore, its maintainability. Even with the best editor, expressing yourself in XML is a lot less productive than coding the same ideas in a notation specifically designed for a given problem. (Try rewriting a simple make file into its Ant XML equivalent.) So, if humans will typically communicate with your software using a language, invest some effort in its design rather than relying on the bland (dis)comfort of XML.

If humans will typically communicate with your software using a language, invest some effort in its design.

Another popular misuse of XML involves thin-wrapping arbitrary data with XML tags. Because XML is flexible, it's easy to take any data format, throw in a few tags in the most convenient places, and (following the letter of the XML definition) call that an XML document. Yet, such documents are difficult to process effectively with standard XML tools. Their validation is a charade, and transformations and queries become all but impossible. For instance, consider the XML file format used for storing iTunes libraries. Its generation apparently takes the shortcut of converting Apple's Core Foundation types into a so-called property list, which looks like XML on the outside. Yet the contents of such files are key/value pairs, such as the following:

```
<key>Name</key><string>Audiobooks</string>
<key>Playlist ID</key><integer>94</integer>
```

In a better, tailor-designed XML file format, we'd expect this pair to be something like

```
<name id="94">Audiobooks</name>
```

A similarly dysfunctional XML file will result if we dump a relational database in XML as columns, rows, and tables. Again, we miss the opportunity to express in XML the deeper relationships between our records, which is really XML's strength.

So, when you're designing an XML document, place yourself in the mindset of its consumer. Think, what's the best possible structure you would expect? Then invest in mapping your data into the schema you've designed.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.

Post your comments online by visiting the column's blog: **www.spinellis.gr/tools**