Don't let delusions about XML develop into a virulent strain of XML fever.

BY ERIK WILDE AND ROBERT J. GLUSHKO

XML Fever

THE EXTENSIBLE MARKUP LANGUAGE (XML), which just celebrated its 10th birthday,⁴ is one of the big success stories of the Web. Apart from basic Web technologies (URIs, HTTP, and HTML) and the advanced scripting driving the Web 2.0 wave, XML is by far the most successful and ubiquitous Web technology. With great power, however, comes great responsibility, so while XML's success is well earned as the first truly universal standard for structured data, it must now deal with numerous problems that have grown up around it. These are not entirely the fault of XML itself, but instead can be attributed to exaggerated claims and ideas of what XML is and what it can do.

This article is about the lessons gleaned from learning XML, from teaching XML, from dealing with overly optimistic assumptions about XML's powers, and from helping XML users in the real world recover from these misconceptions. Shamelessly copying Alex Bell's "Death by UML Fever,"¹ we frame our observations and the root of the problems along with possible cures in terms of different categories and strains of "XML fever." We didn't invent this term, but it embodies many interesting metaphors for understanding the use and abuse of XML, including disease symptoms, infection methods, immunization and preventive measures, and various remedies for treating those suffering from different strains.

XML fever can be acquired in many different ways, but the most prevalent way is to be infected by the idea that XML enables almost magical universal interoperability of information producers and consumers. XML fevers can be classified as basic, intermediate, and advanced:

Basic strains infect XML neophytes, but most of them recover quickly. It can be disappointing to discover that the landscape of XML technologies is not as simple as expected, and that working with the associated tools requires some getting used to, but most people develop some immunity to the XML hype and quickly begin to do useful work with it.

Intermediate strains of XML fever are contracted when XML users move beyond simple applications involving structured information and encounter models of data, documents, or processes. A recurring symptom in these varieties of XML fever is mild paralysis brought on by having to select a schema language to encode a model, trying to choose among the bewildering number of features in some languages, or trying to "round-trip" a model between different environments.

Advanced strains of XML fever often take hold after exposure to the proliferation of more complex and esoteric XML-based technologies layered on top of it. These advanced diseases are

harder to catch, but they are also harder to remedy because people who have caught these advanced strains tend to congregate with others with the same diseases and they are continually reinfecting each other.

Basic Strains

One of our favorite teaching moments is to start an introductory XML lecture with the statement, “XML is a syntax for trees,” and that this is all there is to it, so no further explanation is required. Of course, there is more to it, and we manage to fill a complete course with it, but the essence of XML really is simple and small. This is elegant to us but a disappointment to many XML beginners who expect something bigger and more complicated to match up with all the hype they have heard. In fact, XML’s character-based format lures many XML beginners to assume they can simply use their trusted text-processing tools, which is the inevitable path to the first XML fever:

Parsing pain. At first sight, XML’s syntax looks as if it would be easy to use simple text-processing tools for accessing XML data, so that a “desperate Perl hacker” could implement XML in a weekend. Unfortunately, not all XML documents use the same character encoding; character references must be interpreted; entities must be resolved; and so on... As soon as the output from a wider array of XML producers is considered, it becomes apparent that for robustly parsing XML with text-processing tools, the tools must implement a complete XML parser. This becomes most painfully evident when XML processing needs to take XML Namespaces into account (often leading to an infection with the intermediate namespace nausea fever).

After overcoming parsing pain and starting to use an XML parser, beginners usually understand what we mean when we say that XML is a syntax for trees, but they do not as quickly grasp that XML uses multiple tree models, and depending on which XML technology one is using, the “XML tree” looks slightly different. Thus, the second basic strain of XML fever is:

Tree trauma. This is caused by exposure to XML’s various tree models, such as XML itself, DOM, the Infoset, XPath 1.0, PSVI, and XDM. All of these

tree models share XML’s basic idea of trees of elements, attributes, and text, but have different ways of exposing that model and handling some of the details. In fact, while XML itself explicitly states that XML processors must implement all of XML (apart from validation, the standard has no optional parts, which is a smart thing for a standard to do), some of the more recent tree models exhibit the “extended subset” nature of technologies, which can often lead to incompatibilities among implementations. For example, PSVI—the data model of an XML document validated by an XML Schema (for the rest of the article, we refer to W3C’s language as XSDL)—is based on the Infoset, which is a subset of the full information of an XML document, and extends that subset with information made available by the schema and the validation process.

While XML is available in a number of various “tree flavors,” the W3C has settled (after a very long process) on the Infoset model as the core of many XML technologies. This means it would be technically more accurate to say that most XML technologies available today are actually Infoset technologies. XML has become one way (and so far the only standardized one, but with the upcoming binary Infoset format EXI as a more compact alternative) of representing Infosets. Of course, the W3C does not want to give up the brand name of XML and still calls everything “XML-based.” As a result, XML users can easily get affected by a peculiar ailment:

Infoset ignorance. Instead of XPath, XSLT, and XQuery, these technologies’ proper names would be IPath, ISLT, and IQuery, because they are Infoset-based. Victims of Infoset ignorance take the W3C’s branding of everything as XML at face value and sometimes invest a lot of energy trying to build XML processing pipelines that preserve character references and other markup details. Infoset ignorance prevents its victims from seeing that this approach cannot succeed as long as they are using standards-based tools.

The remedy for Infoset ignorance is to select a set of XML technologies with compatible tree models. This usually also cures tree trauma, because now XML users can focus on a specific variety of XML tree. Depending on the specific

technologies chosen, though, tree trauma can metastasize into a more severe disease caused by failure to appreciate the somewhat obscure ways in which some XML technologies process trees:

Default derangement. Tree trauma can develop into default derangement if XML users are exposed to and experiment with schema languages such as DTDs and XSDL that allow default values. These languages cause XML trees to change based on validation, which means that XML processing is critically based on validation. Because it is often not feasible to quarantine XML users to keep them from these schema languages, a better prescription is to put them on a strict diet of design guidelines to avoid these potentially dangerous features.

Among the core components of virtually all XML scenarios today are XML Namespaces. They are essential for turning XML’s local names into globally unique identifiers, but the specifics of how namespaces can be declared in documents, and the fact that namespace names are URIs that do not need to be dereferenceable, have not yet failed once to confound everybody trying to start using them. A very popular XML fever thus is:

Namespace nausea. No matter how often we try to explain that XML Namespaces have no functionality beyond the simple association of local names and namespace names, many myths and assumptions surround them. For example, many students assume that namespaces must refer to existing resources and ask us how to “call the namespace in a program.” And even though they should be simple, XML is often serialized by tools that do not allow much control over how namespaces are treated, creating XML documents that exhibit various kinds of correct but very confusing ways of using namespaces. A particularly nasty secondary infection caused by namespace nausea can be contracted when using a specific kind of XML vocabulary:

Context cataracts. If QNames (the colon-separated names combining namespace prefixes and local names) are allowed to appear as content of XML documents (such as in attribute values or element content), they make the content context dependent. This means that such XML content can be

correctly interpreted only within its context in the XML document (where all in-scope namespace declarations can be accessed), or it must be decontextualized by parsing it and replacing each QName with a context-independent representation. Unfortunately, no standard exists for this latter approach, which makes this contextualized content brittle and hard to work with.

The strains described so far manifest themselves in basic XML processing tasks. As soon as XML users begin work with business information and processes, they must confront the challenge of understanding what XML structures actually mean. This task exposes them to a dangerous virus encoded in the catchy slogan that XML is “self-describing.”

We could be charitable and assume that when people say XML is self-describing, what they really mean is “compared with something else that clearly isn’t.” The least self-describing information consists of just a stream of alphanumeric characters of some text format, as they might be on a punch card. This delimiter-less encoding does not even make explicit the tokenization of the characters into meaningful values, so there is not any “self” to which any description could be assigned. The possibility of self-description emerges only when we separate the values with commas or some other delimiter character, which tells us what information components must be described. XML goes one step further with the syntactic mechanisms of paired text labels to distinguish the information components in a stream of text and quotes to associate one bit of information as an attribute of another. It is certainly fair to say that XML is on average more self-describing than other text-based encoding syntaxes, but that is like saying the average dwarf is taller than the average baby; neither is tall enough to excel at basketball.

From a more technical perspective, it is also true that XML is self-describing in the limited sense that the data structure (one of the XML trees, see tree trauma) can be reconstructed from an XML document (and maybe its schema, if processing takes place in an environment susceptible to default derangement).

While XML’s success is well earned as the first truly universal standard for structured data, it must now deal with numerous problems that have grown up around it. These are not entirely the fault of XML itself, but instead can be attributed to exaggerated claims and ideas of what XML is and what it can do.

When most people say that XML is self-describing, however, they are being captured by a delusion that this refers to actual semantics, overlooking the fact the XML has almost no predefined semantics (the only exception being one predefined attribute for identifying languages). The disease is most likely caused by the many XML examples that show element and attribute names that seem to be self-describing because they are labeling the syntactic components. It could be prevented with examples that merely show how the XML markup characters distinguish the information being described from the markup that is part of its structural description:

```
<xxx yyy="4567">850</xxx>
<zzz>20060812</zzz>
```

Using syntactic mechanisms to provide clues to the element and attribute semantics is convenient, but this is the cause of a very common strain of XML fever:

Self-description delusion. XML’s ability to define names for elements and attributes, and the widespread assumption that these names have some intrinsic semantics, often cause victims to assume that the semantics of an XML document are self-evident, openly available just by looking at it and understanding the names. Frequently, this strain of XML fever causes great discomfort when the victims learn that XML does not deal with semantics, and that common understanding has to be established through other mechanisms. Victims weakened by self-description delusion are often infected by one or more of the intermediate or advanced strains of XML fever, which promise to easily and permanently cure the pain caused by self-description delusion.

Recovery from self-description delusion can take a great deal of personal commitment and effort. Victims must learn how to define or adapt an XML vocabulary, or to adopt technologies that are explicitly focused on semantics, not just syntax. In either case, these steps risk exposure to strains of XML fever beyond the basic types.

Intermediate Strains

If self-description delusion is appropriately diagnosed and treated, XML users

often recover with improved insight. They now realize that XML's basic technologies and toolset can be employed for basic processing tasks involving structured data, but that most applications involve models of the application data or processes. XML is based on tree structures as the basic model, and this does not always provide the best fit for application-level models, which can cause trouble when mapping these nontree structures to XML:

Tree tremors. Whereas tree trauma (discussed earlier) is a basic strain of XML fever caused by the various flavors of trees in XML technologies, tree tremors are a more serious condition afflicting victims trying to manage data in XML that is not inherently tree-structured. The most common causes are data models requiring nontree graph structures and document models needing overlapping structures. In both cases, mapping these models to XML's tree model results in XML structures that cannot conveniently represent the application-level model.

We often tell students that "the best thing about XML is the ease with which you can create a new vocabulary." But because XML allows well-formed documents (as opposed to valid documents that must conform to some schema), it is actually possible to use vocabularies that have never been explicitly created: documents can simply use elements and attributes that were never declared (let alone defined) anywhere. Well-formedness can be appropriate during prototyping but is reckless during deployment and almost certainly subverts interoperability. Unfortunately, many XML users suffer from a condition that prevents them from seeing these dangers:

Model myopia. Starting from a prototype based on well-formed documents, some developers never bother to develop a schema, let alone a well-defined mapping between such a schema and the application-level data model. In scenarios leading to this condition, validation often is only by eye (key phrases for this technique are "looks good to me" or "our documents usually sort of look like these two examples here"), which makes it impossible to test documents strictly for correctness. Round-trip XML-to-model and reverse transformations cannot be reliably implemented,

and assumptions and hacks get built into systems, which inevitably cause interoperability problems later on.

If model myopia is diagnosed (often by discovering that two implementations do not interoperate correctly because of different sets of assumptions built into these implementations), the key step in curing it is to define a schema so that the XML structures to be used in documents are well defined and can be validated using existing tools. As soon as this happens, the obvious question is which schema language to use. This can be the beginning of another troublesome development:

Schema schizophrenia. DTDs are XML's built-in schema language, but they are limited in their expressiveness and do not support essential XML features (most notably, they do not work well with XML Namespaces). After considering various alternative languages, the W3C eventually settled on XSDL, a rather complex schema language with built-in modeling capabilities. XSDL's expressiveness can directly cause an associated infection, caused by the inability to decide between modeling alternatives:

Schema option paralysis. XSDL's complexity allows a given logical model to be encoded in a plethora of ways (this fever will mutate into an even more serious threat with the upcoming XSDL 1.1, which adds new features that overlap with existing features). A cure for schema option paralysis is to use alternative schema languages with a better separation of concerns (such as limiting itself to grammars and leaving data types and path-based constraints to other languages), most notably RELAX NG.

Using more focused schema languages and targeting a separation of concerns leaves schema developers with a choice of schema languages. In addition, at times it would be ideal to combine schema languages to capture more constraints than any one could enforce on its own. The choice of schema languages, however, is more often determined by available tool support and acquired habits than by a thorough analysis of what would be the most appropriate language.

Since schema schizophrenia (with occasional bouts of schema option paralysis) can be a painful and long-

lasting condition, one tempting way out is not to use schema languages as the normative encoding form for models and instead generate schemas from some more application-oriented modeling environment or tool. Very often, however, these tools have a different built-in bias, and they rarely support document modeling. This causes a very specific problem for generated schemas:

Mixed content crisis. XML's origin as a document representation language gives it capabilities to represent complex document structures, most notably mixed content, essential in publications and other narrative document types. Most non-XML modeling environments and tools, however, are data oriented and lack support for mixed content. These tools produce XML structures that look like table dumps from a relational database, lacking the nuanced document structures that are crucial in a document-processing environment.

Because the approach of generating schemas has the advantage that developers of XML schemas never have to actually write them (or even look at them), it also can be the cause of one of the most troubling XML problems that is often experienced when encountering schemas generated from UML models or spreadsheets:

Generated schema indigestion. More abstract models have to be mapped to XML vocabularies for XML-based information exchange. Most modeling tools and development environments export models to XSDL and use that schema for serializing and parsing instances. Because of the perniciousness of schema schizophrenia, however, this model-to-schema encoding is complex and tool dependent. Generated schema indigestion often afflicts those who try to use the schema or instances outside the context of the tools that generated them. This first contact with generated schemas can be very frustrating and distasteful, because unless the same XML encoding rules are followed in both contexts, XML might not be easy to work with and certainly is neither interoperable nor extensible.

These intermediate strains of XML fever mostly revolve around the problem of how to create and use well-defined descriptions of XML vocabular-

ies. Before we continue to describe the more advanced strains of XML fever that may result from these intermediate fevers and attempts to cure them, it is important to point out that a good way of avoiding them is to reuse existing XML languages, thus avoiding the efforts and risks of inventing something new.

In an online follow-up to “On Language Creation,”³ Tim Bray (one of the creators of XML) says, “If you’re going to be designing a new XML language, first of all, consider not doing it.” This is a very important point, because the ubiquity of XML makes it likely that for any given problem, somebody else might have already encountered it and solved it. Or for a given problem, it might be possible to divide it into smaller parts or to map it to a more general problem, and to find existing solutions for these.

Of course, there is a chance that no prior work exists or that the available solutions are unsatisfactory, but it really is worth the effort to evaluate existing solutions because a vocabulary can represent hundreds or even thousands of hours of analysis and encoding. For example, the Universal Business Language (UBL), a set of information building blocks common to business transactions and several dozen standard documents that reuse them, is the result of years of work by numerous XML and business experts—and the UBL effort itself began in 2001 by building on the XML Common Business Library (xCBL), on which work began in 1997.

We always tell students the worst thing about XML is the same as the best thing: the ease with which you can create a new vocabulary. Language design is fundamentally hard, but XML has made it deceptively simple by lowering the syntactic threshold. The conceptual tasks of creating shared vocabularies that are globally understood, well defined in every necessary respect, and reasonably easy to use have not been made easier by XML. XML has just given us a good toolset to describe and work with these languages once we have them, but defining them still is hard work.

This, of course, is not a secret to computer scientists, and the fact that XML has no semantics when they are essential to meaningful information

We always tell students the worst thing about XML is the same as the best thing: the ease with which you can create a new vocabulary.

exchange led to the idea of the Semantic Web.² The value proposition of the Semantic Web is compelling: a common way of representing semantics makes it easier to express, understand, exchange, share, merge, and agree on them. The Semantic Web, however, is also the leading cause of the more advanced strains of XML fever.

Advanced Strains

If semantics are important, and since an XML schema defines only structures (that is, syntax), then semantics must be specified in some other way. This can happen informally by prose describing the meaning of the individual components and parts of a schema, or more formally, by using some model for specifying semantics. The Semantic Web is the most popular candidate for such an environment; it is based on a model for making statements about resources, the Resource Description Framework (RDF), with various technologies layered on top of that, such as those for describing schemas for RDF.

One important observation about the Semantic Web that is often missed is that it introduces not only models for semantics (various schema languages for RDF), but also a new data model, which means that XML’s tree structures are no longer the core data structures for representing data. RDF can be expressed in XML, but there are many different ways of doing it, which can cause a very specific illness:

RDF rage. RDF’s most widely used syntax is XML based, but there are many different ways in which the same set of RDF triples can be expressed as XML, so working with RDF data is almost impossible using basic XML tools, even for simple tasks such as comparing RDF data. This inability to use a seemingly related toolset for a seemingly related task often is the first symptom through which XML users learn that they are now suffering from more advanced strains of XML fever.

In a more classical view of information organization, the meaning of terms can be specified in a variety of ways. Ordered by complexity, popular approaches are controlled vocabularies, taxonomies, thesauri, and ontologies. RDF can be used to implement any of these concepts, but RDF schemas are most often referred to as on-

tologies. This is in part a result of free standards-based tools for creating ontologies such as Protégé and SWOOP; just as we mentioned with schema option paralysis, the availability of tools shapes the languages people use and the choices they make. The relative unfamiliarity and the vague “hipness” of the “ontology” world, however, can give XML users anxiety about their ability to adjust to the RDF/OWL world with more rigorous semantics. As a result, they often overcompensate:

Ontology overkill. Operating in an environment that focuses on semantics, victims of ontology overkill tend to overmodel semantics, creating abstractions and associations that are of little value to the application but make the model much harder to understand and use. Ontology overkill forces its sufferers not only to overmodel, but also often to fail at doing so, because it is much harder to define an ontology (in its fullest sense), and to identify, understand, and validate all its implications, than it is to define a controlled vocabulary.

If XML fever sufferers come in contact with communities where Semantic Web ideas are widespread and well established, they quickly discover that most of their knowledge acquired in the basic and intermediate phases of the XML learning curve does not apply anymore. The reason for this is that the Semantic Web creates a completely self-contained world on top of other Web technologies, with the only intersection being the fact that resources are identified by URI. As a result, Semantic Web users become blissfully unaware that the Web may have solutions for them or that there could be a simpler way of solving problems. Seeing the Semantic Web as the logical next step of the Web’s evolution, we can observe the following condition:

Web blindness. This is a condition in which the victim settles into the Semantic Web to a degree where the non-Semantic Web does not even exist anymore. In the pure Semantic Web, lower-level technologies no longer need to evolve, because every problem can be solved on the semantic layer. Web blindness victims often are only dimly aware that many problems in the real world are and most likely will be solved with technologies other than

Semantic Web technologies.

If victims of Web blindness have adjusted to their new environment of abundant RDF and start embracing the new world, they may come in contact with applications that have aggregated large sets of RDF data. While RDF triples are a seemingly simple concept, the true power of RDF lies in the fact that these triples are combined to form interconnected graphs of statements about things, and statements about statements, which quickly makes it impossible to use this dataset without specialized tools. These tools require specialized data storage and specialized languages for accessing these stores. Handling these large sets of data is the leading cause of an RDF-specific ailment:

Triple shock. While RDF itself is simple, large datasets easily contain millions of triples (for truly large datasets this can go up to billions), and managing and querying such a big dataset can become a considerable challenge. If the schema of these large datasets is simple, but ontology overkill has set in and it has been reformulated as an ontology, handling this dataset may become considerably harder, without any immediate benefit.

Semantic Web technologies may be the correct choice for projects requiring fully developed ontologies, but Semantic Web technologies have little to do with the plain Web and XML. This means that neither should be regarded as a cure for basic or intermediate XML fevers, and that each has its own set of issues, which are only partially listed here.

The Prescription

We probably cannot prevent these varieties of XML fever, especially the basic strains, because it is undoubtedly a result of the hype and overbroad claims for XML that many people try it in the first place. We can do a better job of inoculating XML novices and users against the intermediate and advanced strains, however, by teaching them that the appropriate use of XML technologies depends on the nature and scope of the problems to which they are applied. Heavyweight XML specifications such as those developed by OASIS, OMG, and other standards organizations are necessary to build robust enterprise-class XML applications, and

Semantic Web concepts and tools are prerequisites for knowledge-intensive computation, but more lightweight approaches for structuring and classifying information such as microformats will do in other contexts.

When someone first learns about it, XML may seem like the hammer in the cliché about everything looking like a nail. Those of us who teach XML, write about it, or help others become effective users of it, however, can encourage a more nuanced view of XML tools and technologies that portrays them as a set of hammers of different sizes, with a variety of grips, heads, and claws. We need to point out that not everyone needs a complete set of hammers, but information architects should know how to select the appropriate hammer for the kind of hammering they need to do. And we should always remember that pounding nails is only one of the tasks involved in design and construction.

XML has succeeded beyond the wildest expectations as a convenient format for encoding information in an open and easily computable fashion. But it is just a format, and the difficult work of analysis and modeling information has not and will never go away. ■

References

1. Bell, A.E. Death by UML fever. *ACM Queue* 2, 1 (Mar. 2004), 72-80.
2. Berners-Lee, T., Hendler, J.A., Lassila, O. The Semantic Web. *Scientific American* 284, 5 (May 2001), 34-43.
3. Bray, T. On language creation. In *Proceedings of XML 2005* (Atlanta, GA, Nov. 2005).
4. Bray, T., Paoli, J., Michael Sperberg-McQueen, C. Extensible markup language (XML) 1.0. World Wide Web Consortium, Recommendation REC-xml-19980210 (Feb. 1998).

Erik Wilde (dret@berkeley.edu) is a visiting assistant professor in the School of Information at the University of California at Berkeley, where he is also technical director of the Information and Service Design program.

Robert J. Glushko (glushko@ischool.berkeley.edu) is an adjunct professor at the University of California at Berkeley in the School of Information, the director of the Center for Document Engineering, and one of the founding faculty members of the Information and Service Design program.

© 2008 ACM 0001-0782/08/0700 \$5.00