On Language Creation

Tim **Bray**

November 2005.

Abstract

Next year, the notion of building your own markup language for your own application, while still conforming to a standard, will be 20 years old. During that twenty years, a huge number of custom languages - at least hundreds, perhaps a couple of thousand - have been attempted. Almost all have been miserable failures. That is to say, the vast majority have failed to achieve wide adoption, and those that were adopted have often failed to achieve their goals, whether of reducing costs, enriching applications, or both.

This paper examines this history of failure and draws conclusions about the decision process as to whether to proceed with language design, and, in the case that the design of a new language is undertaken, the trade-offs that obtain during that process.

Table of Contents

1. Introduction

As of mid-2005, in excess of six hundred distinct XML-based languages had made sufficient impression on the world to have been noted at Robin Cover's XML Cover Pages [http://xml.coverpages.org/xmlApplications.html].

How many of these languages can be considered successful; that is to say, met the objectives of those who set out to create them? The answer is clearly less than 100%; in fact, community experience suggests that the failures outnumber the successes by a considerable margin.

Given that the invention of an XML-based language involves a substantial amount of work, and that that work is often political, tedious, and unpleasant, this substantial number of forgotten languages has to be seen as something of a tragedy.

The obvious conclusion—that language design is a high-risk activity which should not be undertaken lightly—is hard to resist. Sometimes, though, whether or not it's a good idea, people do choose to engage in designing XML-based languages; the past two decades of experience have lessons to teach on how to minimize the risk of failure.

2. Why Not to Design a Language

It is generally desirable to avoid activities that are costly, time-consuming, difficult, and have a high risk of failure. The design of new XML-based languages not only has all these characteristics, it is arguably actively harmful in the context of widely-applicable business goals.

2.1. Network Effects

XML provides interoperability at the level of syntax, expressing all documents as nested sequences of labeled pieces of text. Such interoperability is cheap to achieve and delivers correspondingly little value at the business level.

Real interoperability is a function of shared *semantics*, not syntax. The most obvious example in all our lifetimes is the World Wide Web; the notion of a hypertext extending across a network was far from new in the early Nineties, but such systems had never achieved wide deployment. One of the key ingredients of the Web's success was the worldwide agreement to standardize on the use of HTML. The facts that HTML is sloppy, has an impoverished repertoire of presentational semantics, and generally disdains the use of hierarchy might have been seen as obstacles, but the advantages of shared semantics proved decisive.

The use of a shared language carries many "network-effect" benefits, but the most obvious is the development of a lively software ecosystem. Since the semantics of HTML were not tied to any particular implementation or application, the rise of link followers, search engines and alternative rendering technologies happened swiftly and on an immensely large scale.

2.2. Software Costs

The success of a newly-create XML-based language requires a substantial investment in software, some of it high-risk.

2.2.1. Authoring

Some (but not all) XML-based languages are intended for human authoring. This is a hard problem. The return on investment in creating authoring environments for custom XML languages is at best questionable, evidenced by the consistent failure of companies delivering such technology to grow beyond start-up scale, or in most cases even to survive.

One issue is cost; the amount of customization required to turn a configurable XML authoring tool into a productive environment for any particular language is typically very high. Even for a widely-used language like HTML, it took the authoring tools a decade or more to become really satisfactory, and the current explosion of the blogosphere has cast a cruel light on the quality and availability of Web authoring software even today.

Obviously, the smaller the number of users a new language will have, the more difficult it is to achieve good return on any investment in authoring tools.

There are some exceptions; for example, I am now writing the text that you are now reading, in a language to which I have little or no exposure, and still achieving a satisfactory degree of productivity based on the combination of the GNU Emacs editing system and the nXML editing support framework. This, however, is a technology combination that would be deployed and used mostly (perhaps only) by hard-core computer geeks.

2.2.2. Validation

XML texts generally need validation, where validation should be read as meaning "automated verification that this text meets the needs of the application that wishes to use it." Obviously, this is a much larger problem than validation according to any schema in any schema language.

Schemas generally cannot check whether the part number allegedly represented by an attribute value actually corresponds to a database record, or whether a library catalogue record contains a valid Library of Congress subject heading.

To be useful, validation needs to be highly sensitive to the needs of some particular application. This means, among other things, given the general-purpose utility of many XML-based languages, that validation needs to be plural rather than singular; multiple different validation protocols will likely need to be applied at different times and places.

2.2.3. Application Function

It is very unlikely that any XML-based language will meet its goals of the available software is limited to authoring and validation. Presumably there is some useful application effect that is the goal of the whole effort, and the software to accomplish that goal, using the XML-based language as input, will need to be written and tested.

2.3. Human Costs

In the deployment of any custom XML-based language, unless the software tool suite is unusually good, it will be impossible to insulate the users from the syntax and semantics of the language. This, for most people, will be a substantial effort. As with many other aspects of custom XML-based languages, a positive return on this investment can be quite difficult to achieve.

3. Minimizing Risk in Language Design

It is clear that, despite the high risks, strenuous effort, and questionable benefits involved in creating a new XML-based language, people persist in undertaking this exercise. In the scenario where the design of a new XML-based language is proceeding, what lessons can we take away from the two decades or so of experience in this area that will minimize the risks and costs?

3.1. Model vs. Syntax

Many of the people who become engaged in XML-based language design have been trained in Computer Science and may be professional programmers or system architects. Such people are accustomed to constructing abstract, formal models as a basis for forward progress; these models eventually finding expression in executable code constructs, relational schemas, and the like. The practice of modeling is sufficiently widespread that there is an industry of supporting tools and languages, such as UML in the object-oriented community and OWL in the knowledge-representation community.

On the other hand, those computer professionals who focus on network protocols tend to have an obsessive focus on the syntax, the "bits on the wire", feeling that nothing else really has any objective reality.

The community of people who consider themselves "document-centric", mostly inhabiting the publishing professions, have been observed, perhaps surprisingly, to exhibit similar divided perceptions of reality.

Communication between model-centric and syntax-centric people is difficult and in some cases impossible.

The author of this paper is strongly syntax-centric, but has observed that the application of formal models can confer benefits on document design. For example, consider Mark Pilgrim's discussion [http://www.xml.com/pub/a/2003/08/20/dive.html] of the benefits of applying RDF thinking to the design of Atom.

On the other hand, model-centric people should be prepared to acknowledge that the design of syntax is far from ephemeral, and it is clearly not the case that a satisfactory syntax design will "automatically fall out" of a good modeling exercise.

Finally, it should be strongly born in mind that for most markup languages, it is a basic expectation that they will be will be processed by multiple independent software implementations created by multiple independent groups who may have very different perceptions of reality. It is, after all, a central benefit of XML that it makes this possible.

Thus, it should normally be expected that a single syntax will be processed based on multiple incompatible data models. For example, the data models used by a browser, a link follower, and a full-text indexer processing the same Web page are not expected to be identical or in some cases even compatible; and this is a virtue, not a problem.

3.2. Semantic Gaps

A core reason for using XML is to enable the exchange of data between different applications running in different computing environments. It is always the case that there will be semantic gaps between the perceptions of reality in these environments. For those who have worked on B2B vocabularies, the degree to which seemingly-simple concepts such as "ship date" and "discount" stand for radically different things is an unending source of unpleasant surprises.

One of the great computer programmers, the late Phil Karlton [http://karlton.hamilton.com/], said "There are only two hard things in Computer Science: cache invalidation and naming things". Language design is all about finding names for things, and this is cruelly effective at exposing semantic gaps.

Since there is no way to eliminate this problem, the only useful way to fight it is to minimize it, by doing less. If every tag or attribute in your language represents a potentially-dangerous semantic compromise, then the fewer tags and attributes, the fewer sources of trouble for users and implementors, and the greater the chances for semantic interoperability.

3.3. Time to Market

Language design is inherently time-consuming; not least because it is usually done in a committee or working group, one of the least efficient forms of human organization. Discussions of what some markup construct means, and no less, what it should be called, are observed consistently to expand to fill all the time available.

This is a source of problems, since most language-design efforts do not begin until there is a perceived need for the output; which need must necessarily go unmet until the process completes. In fact, it's worse than that, because the application software which will be deployed to make use of the new language can't really be implemented until the language is stable.

While this is another hard problem, there are some well-known management practices which can ameliorate it. Language-design efforts tend to benefit from small working groups, strong chairs, realistic goals, rigorous attention to deadlines, and a willingness to sacrifice features to meet ship dates.

3.4. Dynamism vs. Stability

Many, perhaps most, language specifications are versioned. No sooner does the working group release 1.0 than they turn to work on 1.1 or 2.0, and there are teams that have continued producing releases of language specifications, achieving double-digit release numbers over a period of decades.

This practice is actively harmful and should, if at all possible, be avoided. A release of a language specification typically requires expensive re-engineering and re-implementation of the authoring, validation, and application software.

Languages which are stable are far more likely to serve as platforms for the develop of the software ecosystems which are perhaps the single greatest discriminant of success from failure for a markup language.

Examples would include RSS 2.0, HTML after about 1997, and to some extent, XML itself. The desire to produce a version 2.0 is widespread among language designers, because very few 1.0 releases solve the whole problem or are free from lacunae and outright errors. Extensibility (see below) provides some degree of consolation, but history should have taught us that the benefits of language stability, in most cases, exceed the benefits of producing successive revisions.

3.5. Extensibility

Given the benefits, already examined here, of producing a language that is small and stable as opposed to large and growing, and given also the fact that the world changes and perceptions evolve, extensibility in an XML-based language is an unambiguously good thing. Fortunately, recent years have been a rich source of lessons in how to go about achieving this.

3.5.1. Namespaces

Years after their invention, XML Namespaces remain controversial and undeniably represent a source of clutter and complexity in XML instances. Having said that, they do provide a fairly bullet-proof way to extend XML vocabularies. Many modern language specifications contain statements along the lines of "This element can contain any markup whatsoever in a namespace other than its own"; and programmers find such instructions easy and straightforward to deal with.

3.5.2. Extension Points

Some languages allow extensions (usually from other namespaces) to appear more or less anywhere in a document; others document specific extension points and bless the appearance of "foreign markup" only in a list of designated elements.

It is not obvious whether either approach is better, or even what all the trade-offs are, but this is clearly an issue to which attention should be paid.

3.5.3. Must-understand and Must-ignore

One of the acknowledged reasons for the success of the Web was what has been called the "implicit must-ignore rule". Web software implementations historically have had a policy of, when they encounter that they don't recognize, silently ignoring it and proceeding with their work. In the specific case where markup appears in a context where text is being displayed, the policy is generally to discard the markup and the attributes, and display the character data that appears in element content.

Modern language designers should take this lesson from the Web to heart; the default assumption should generally be that a must-ignore rule is in effect, and to avoid ambiguity, this should be written into the language specification. The rule is simple: applications are not allowed to fail to function as a result of the occurrence of unrecognized markup. It is a matter of judgment as to whether the specific Web behavior of displaying text in unknown markup's element content is appropriate for other applications.

On the other hand, there are other situations, particularly security-related, where it is not appropriate or safe to ignore unrecognized markup. In these situations, it is appropriate to include a "must-understand" signal in a language design. This can take the form, for example, of an attribute designed into the language, which foreign markup can provide to indicate that the document cannot be processed by software which does not understand that markup. Other schemes, such as a list of XML namespaces which must be recognized, are plausible ways to implement must-understand.

The implementation of must-understand in SOAP is generally regarded as a problem; it is potentially necessary to parse through a high volume of content before arriving at a signal that the already-parsed content must be ignored. Clearly this makes life difficult for implementors.

In any case, the inclusion of a must-understand capability in a language necessarily makes it to some degree less extensible, and to some degree a more complex challenge for implementors. Leaving out must-understand, where possible, is thus desirable.

4. Conclusions

The primary conclusion is that XML-based language design is an activity best avoided; not only because it is expensive and dangerous, but because it may in many cases be actively harmful. If there is an alternative, usually an existing language which can plausibly be repurposed, that alternative should be given a chance if at all possible.

Having said that, there are some best practices that have emerged from our decades of experience which can reduce the risk and cost of language design. To summarize:

1. Accept that there will be a clash between the model-centric and syntax-centric world-views, but bear in mind that successful XML-based languages support the use of multiple software implementations that cannot be expected to share a data model.

2. The process of formally itemizing and naming the objects to be described by a language can be expected to expose semantic gaps in the human understanding of these objects. Since this is inevitable, the only way to minimize the program is to specify the minimum possible number of objects.

3. Designing languages is time-consuming and subject to schedule overrun; this should be addressed using standard process management techniques, among the most effective of which is minimizing the size of the design group.

4. Languages which, once specified, do not change, are maximally friendly to computer programmers and thus more apt to develop a supporting software ecosystem. Thus, if at all possible it's best to ship 1.0 of a language specification and then disband the working group that created it.

5. Languages should be extensible, and this extensibility should wherever possible be based on the must-ignore principle. A must-understand capability is sometimes necessary but increases fragility and cost.

Acknowledgements

This paper is entirely reliant on, and is thus dedicated with grateful acknowledgments to, all those who have laboured so hard and so long over the years to produce today's rich menagerie of XML dialects; it only relates the lessons from their experience. Specific thanks are due to James Clark for the wonderful NXML Emacs editing mode.

Biography

Tim **Bray**
Director of Web Technologies
Sun Microsystems, Inc. [http://www.sun.com]
Vancouver
British Columbia
Canada
http://www.tbray.org/ongoing/ [http://www.laurenwood.org]

Tim Bray has been working with descriptively-marked-up text, sans DTDs, since 1987. He blogs at ongoing [http://www.tbray.org/ongoing].