XML

<and Semi-Structured Data>

XML, as defined by the World Wide Web Consortium in 1998, is a method of marking up a document or character stream to identify structural or other units within the data. XML makes several contributions to solving the problem of semi-structured data, the term database theorists use to denote data that exhibits any of the following characteristics:

- Numerous repeating fields and structures in a naive hierarchical representation of the data, which lead to large numbers of tables in a second- or third-normal form representation
- Wide variation in structure
- Sparse tables

XML provides a natural representation for hierarchical structures and repeating fields or structures. Further, XML document type definitions (DTDs) and schemas allow fine-grained control over how much variation to allow in the data: Vocabulary designers can require XML data to be perfectly regular, or they can allow a little variation, or a lot. In the extreme case, an XML vocabulary can effectively say that there are no rules at all beyond those required of all well-formed XML. Because XML syntax records only what is present, not everything that might be present, sparse data does not make the XML representation awkward; XML storage systems are typically built to handle sparse data gracefully.

The most important contribution XML makes to the problem of semi-structured data, however, is to call into question the nature and existence of the problem. As the description makes clear, semi-structured data is just data that does not fit neatly into the relational model. Referring to “the problem of semi-structured data” suggests subliminally that the problem lies in the failure of the data to live up fully to the relational model, rather than in the model and its failure fully to support the natural structure of the data.

In the wild (that is, in documents, reports, and program data structures as they are encountered in daily life), information takes forms rather different from third normal form. XML arose from efforts to represent documents in a device- and application-independent way, and it reflects the complexity of documents and their stubborn refusal to fit into tabular form.

In XML, data can have an elaborate and intricate structure that is significantly richer and more complex than a table of rows and columns. Calling this semi-structured is misleading, just as it would be to describe DNA molecules as semi-structured because they are less simply regular than those of table salt. XML seeks to make possible capturing and expressing the structure of the data as we understand it, without forcing it into a too-simple structure.

C. M. SPERBERG-MCQUEEN, WORLD WIDE WEB CONSORTIUM

XML

<and Semi-Structured Data>

Defining what counts as a valid document requires a definition language richer than that provided by relational systems. In a typical technical manual, lists and paragraphs may appear at the same level, but some documentation departments require that at least two paragraphs must appear between any two lists. This is easy to define using regular expressions or similar tools from formal language theory; it is not straightforward to define using mechanisms such as those of SQL schemas.

XML DATA MODELS

As with any meta-language, of course, XML has an effect on the semantics of its applications. XML invites us first of all to consider information as having *structure*. This distinguishes it not from database systems, which have always emphasized the fine-grained structure of information, but from contemporary document processing, which frequently fails to allow the meaningful exploitation of document structure. Good XML design places so much emphasis on information structure that some partisans of descriptive markup are surprised and amused when they hear that database theorists describe documents as semi-structured.

More specifically, XML invites us to model the structure of our information with elements that form a tree structure, attributes that decorate the nodes of the tree, and inter-nodal links that allow us to model arbitrary graphs, not just trees. For this tree structure, XML provides a straightforward linear representation in the form of a labeled bracketing, which can be used for serial transfer of information. Fundamentally, XML is simply a labeled bracketing in which every element is labeled both at its beginning, with a start-tag, and at its end, with an end-tag.

The document grammar provided by a DTD allows a natural method of expressing constraints on the shape of the tree. The tree structure instantiated by the elements of an XML document can readily be interpreted as the abstract syntax tree for documents conforming to the document grammar.

The data structure, the serial form, and the document grammar complement and reinforce each other in ways that competing methods for representing documents lack. The interplay among these three aspects of SGML (Standard Generalized Markup Language) and XML helps explain why they feel so right to many practitioners. Users of relational database systems benefit from a similar mutual reinforcement of the basic tabular data model and the constraint mechanisms expressible in relational schemata. As a group, however, they have not developed the same kind of fierce affection for and attachment to comma-separated value dumps as SGML and XML users exhibit. Perhaps it's because tabular data has such a simple structure that serializing it does not seem a difficult problem; serializing trees is not hard, but it's difficult enough to make a hard-pressed programmer grateful for an off-the-shelf solution.

One thing may strike some observers as missing from XML, though: The XML specification defines no semantics for XML markup. This is no accident. The desire to support document reuse led the designers of XML to avoid providing any set of formatting primitives. Instead, good XML design emphasizes declarative meaning and tagging the logical structure of the document rather than its appearance in a particular style. The desire to support any kind of document, coupled with an intimate understanding of how variable documents are in practice, led further: XML refrains from defining not only primitives that are format-related, but also any semantic primitives *at all*. The fundamental semantics of an XML application are supplied not by the XML specification but by the designer of the vocabulary. The designer has carte blanche to identify the units of taggable information at any desired granularity and to supply for them any semantics desired.

Since SGML and XML provide no semantic primitives in terms of which documents can be interpreted, they are frequently described as being "just syntax." One may wonder whether a purely syntactic notation constitutes a real step forward. Does defining a labeled bracketing, a tree structure, and a formalism for document grammars really suffice to make XML interesting or important? In some ways, of course, the answer is no; none of these is a particularly complex problem. What graduate student in computer science would regard the problem of developing a notation for serializing trees as requiring more than a weekend's worth of work? Any competent programmer can write a program to parse any reasonably clean notation. Constraint checking is a bit more difficult, but if constraints are checked in a Turing-complete program-

ming language, we may have a better chance of actually expressing all the constraints we would like to express.

In many other ways, however, the answer is yes; purely syntactic notation is a big step forward. XML is interesting to people who wish to exploit their data, because it provides enough commonality among documents to make possible the construction of generic tools instead of tools specialized for single notations. For those who want to spend their time working with their data, rather than building new tools, such generic tools are a great step forward. Generic syntax-directed editors, guided by nothing more complicated than a schema or DTD, make it easier to create good data. Generic tools for data validation make checking for many errors possible, and allow programmers to spend more time on processing the data and less time on checking that the input is clean. Generic browsers and display engines, supported by good style-sheet languages, make it possible to display data in multiple styles tailored for different users or processes.

Today, virtually every modern Web browser can display native XML using CSS (cascading style sheets), and the best browsers also support XSLT (Extensible Stylesheet Language Transformation). Generic tools for data manipulation provide high-level access to the information structures being processed. A well-designed XML vocabulary maps domain-specific structures into elements and attributes, with the corollary that domain-specific processes can readily be described in terms of operations on elements and attributes. An XML-aware tool can become, in effect, a higher-level programming language.

Because the core semantics of an XML document rely not on particular application software but on declarative semantics that are (or should be) explicitly documented, the use of XML really does help ensure data longevity and reusability. Sometimes a rather thin, syntax-oriented, semantically vacuous layer of commonality is all that is needed to simplify things dramatically.

By providing **a common notation for all data**, XML offers the hope of ending this segregation of tabular data from other kinds of data.

THE XML PROCESSING STACK/XML DATA LIFECYCLE

As noted earlier, XML invites us to model information as a tree, but it need not be processed in that form. XML can be understood, and processed, at several different levels of abstraction:

- As a character stream (this is the layer actually defined by the XML spec itself)
- As a sequence of data characters interspersed with markup (a regular language)
- As a tree in the obvious way, with one node per element, and the attributes as decorations on the nodes
- As a graph in which internodal links are defined by parent-child relations between XML elements, by ID/IDREF links, or by application-specific methods of linking between elements
- As a tree or graph annotated with information about data types and validity (as the output of schema validation)
- As an instance of an application data structure, with arbitrary structure, built on the basis of the XML input.

The multiplicity of levels sometimes confuses new users and may affront the sensibilities of those who see virtue only in the definition of a model with access restricted to a single abstraction layer, provided through a well-controlled application programming interface. Users of descriptive markup, however, have been reluctant to limit themselves to a single level of abstraction. It is convenient to have tools that understand the higher-level abstractions in the list above, but it's also convenient to be able to copy or transcode XML documents reliably with mechanisms that understand the data only as a stream of characters. Different levels of abstraction are suitable for different processes and applications: A copy program can be happy with XML-as-character-stream, but a typical application program will typically wish to abstract away from the details of the character stream (for example, silently omitting nonsignificant white spaces). Most XML-aware software prefers to work at the element level, but XML editors are frequently an exception, since users who carefully pretty-print their XML comments or markup in an ASCII editor are typically irate if an XML editor reformats the markup.

If different applications want or need different levels of abstraction, it is certainly understandable that the proponents of descriptive markup, fighting as they were for application independence, would find it an advantage, rather than a disadvantage, to allow XML to be viewed at so many different levels.

Different layers in the XML stack provide access to these different layers of abstraction. The XML data stream

XML

<and Semi-Structured Data>

is provided by the file system, or the network interface, just as for any other notation. XML's stream-internal character encoding declarations provide the information needed to translate from the octet stream actually coming in into a stream of characters. An XML processor reads the octet stream, translates it into characters, and parses it into markup and content, providing access to downstream consumers by means of an interface such as SAX (Simple API for XML) or DOM (Document Object Model). If the resulting output is too high-level (because the application really needs to pay attention to things like white space inside tags in the input), regular expressions can be used in a straightforward way to distinguish markup from content in the input.¹

Most consumers of XML data, however, will read it through a SAX or DOM filter, or through marshaling and de-marshaling code generated from a schema by a data-binding tool. Once the data is represented in data structures native to the programming language, processing can proceed normally.

In a growing number of situations, however, after the data is processed, the results need to be written out in XML form. Once a data owner has concluded that XML provides a good method for ensuring the reusability and longevity of information, it's a short step to the conclusion that *all* the information we care about should be represented in XML, including the output from our programs. Applied consistently, this line of thinking results in an information-processing architecture in which virtually every process becomes a transformation from one XML form to another.

At this point, the impedance mismatch between XML and conventional programming languages can become painful. Languages that treat XML as a native data format and thus eliminate the impedance mismatch can offer a great deal of convenience. Of such languages, the two most prominent by far are now XSLT, the transformations part of XSL (Extensible Stylesheet Language); and XQuery, a new language not yet finally standardized,

although increasingly supported by commercial products from major database vendors.

SOME CONSEQUENCES AND SOURCES OF TENSION

Some astute observers believe that one of the most attractive aspects of XML is its ability to represent both the kind of data that has historically been managed in database management systems and the kind of mostly textual data that has historically eluded database systems, in large part because its structure was too variable to lend itself to representation in databases. By providing a common notation for all data, XML offers the hope of ending this segregation of tabular data from other kinds of data.

Such aspirations raise the stakes for XML, which must work not only in the contexts and for the kinds of information its designers had in mind, but also in other radically different contexts, with very different kinds of data. The result is confrontation of data and human minds coming from very different places. Some specific sources of confrontation and confusion may be worth sketching here.

Grammars and their discontents. Perhaps there is no XML-related topic on which views diverge so widely as on the proper role of the document grammar or schema. In SGML, the document type definition is a required part of the document: If a data stream has no DTD, it is not SGML. Since SGML allows for the omission of start- and end-tags under certain conditions, the DTD is in any case essential to enable the parser to identify the boundaries of elements. Experience shows, however, that some generic applications (e.g., full-text indexing or document formatting or display) make little or no use of the DTD. Some non-SGML markup languages show that minor syntactic changes to the reference-concrete syntax of SGML can make it possible to identify element boundaries reliably without a DTD.

So the designers of XML made the DTD optional.

Many vocabularies are now developed, documented, and deployed without having any formal document grammar at all. Some people see this as a key advantage of XML, since it allows developers to bypass one of the costliest and most difficult stages of application development in the relational world: the development of the schema. Frequently schema development is difficult more for political or social than for technical reasons, and the consensus that ultimately forms around a particular schema may be disturbed by any major disruptive change—including the deployment of the system being designed and the consequent ready availability of information that was hard to get at before. From this point of view, starting

development without a schema—although it may look like setting off on a journey without a map—can make good sense as the beginning of an iterative process of rapid deployment and revision.

Some observers, however, are horrified by the possibility of developing and deploying XML applications without the discipline of having everyone concerned sit down in a room and agree on a schema. (This group includes some of the most prominent of database practitioners and researchers. XML, these researchers complain, allows systems to be deployed in which, for example, human resources departments in different countries of a multinational corporation lacked any common definition of the concept of a full-time employee. When it is pointed out that this can happen in relational systems, too, the response is: “But XML makes it *easy*.”)

Many developers of complex XML-based systems, although not horrified by the DTD being optional, regard it as purely academic. Even though software need not fail with an error message in the absence of a DTD, the fact remains (these developers maintain) that document grammars are absolutely essential to the development of coherent complex applications. They constrain the data, so that software to process the data can focus on handling valid data correctly and can spend less effort on detecting basic syntax errors; the application has less need for defensive programming. Document grammars serve as contracts (in some cases, literally so) between data sources and data consumers—the former committing to provide data conforming to the grammar, the latter committing to handle any data that conforms. In cases of disputes between vendors of gasoline pumps and cash registers over who is responsible for messages being dropped, the messages in question can be validated against the document grammar to adjudicate the question: if the message is invalid, the sender must repair its message generation code; and if the message is valid, the receiver must do the work.

Because they define sets of documents smaller, and with more in common, than the set of all well-formed XML documents, document grammars are useful in reasoning about data and collections of data. As in relational systems, schemas play a key role in allowing database software to reason about possible optimizations of a query.

Similarly, document grammars (as other grammars) can serve to document a model of data, or to explicate a particular concept. Some system designers will specify document grammars for all intermediate forms of the XML in a multistage pipeline, even if they don’t antici-

pate validating the data at that point. The grammar documents the expected output of one stage, and the expected input for the next stage, of processing.

In XML Schema, the document grammar also associates a type with each element and attribute in a document; the result of validation against the grammar is an annotated XML tree with type annotations attached to (valid) nodes.

In some cases, the document grammar simply serves to provide useful names for complex structures, to simplify working with them. An early account of the use of grammars for such purposes was provided by Gaston H. Gonnet and Franklin William Tompa in 1987.²

What is not always appreciated in discussions of the utility or futility of document grammars is that they

For less well-understood data, or data with less stringent requirements, looser rules can be defined.

can vary greatly in the degree of freedom they leave to the data. For well-understood or tightly constrained data, a very tight grammar can be written. Every section must have a head and a subhead. Between any two lists, there must be at least two paragraphs. No section may be shorter than five paragraphs, and any section longer than 20 paragraphs must be divided into subsections. In a structure with 20 subordinate parts, the parts must occur in a particular order.

For less well-understood data, or data with less stringent requirements, looser rules can be defined. In the extreme case, a formal grammar can be written that accepts absolutely any well-formed XML document. Such a grammar does not do much good for validating the input, but it can nevertheless be useful to document explicitly the important fact that at a particular point in a data flow there are no constraints at all beyond the requirement that the data be well-formed XML. (This is often called *Waterloo grammar* in honor of the *New Oxford English Dictionary* project at the University of Waterloo, Ontario, which used an SGML-like tagging system but

XML

<and Semi-Structured Data>

did not formulate a document grammar. The implicit rule was that anything could go anywhere; Waterloo grammar is one that specifies that rule explicitly. Note that the dictionary project did not actually use Waterloo grammars; although useful research was done on grammar induction, the production software developed in the projects used no grammars at all.)

XML and the object-oriented model. The hierarchical structure of XML documents has an obvious similarity to the hierarchical structure of objects. Applications of object technology to XML and vice versa have long been topics of interest. There are obvious parallels between object-oriented design and XML, but several discrepancies complicate life for those straddling this boundary:

- Objects provide for reuse in part by hiding implementation *and data* behind opaque interfaces. XML documents provide for reuse by identifying structures using declarative semantics, but nothing is hidden at all. Many markup theorists take the goal of all markup to be to *exhibit* the structure of the information. Hiding it does not come into question.
- Class hierarchies capture important relations among object classes. Quite often, these are construed as super-set/subset relations: If class *C* has members *x* and *y*, and class *D* extends *C* by adding member *z*, we can think of class *C* as the set of objects with both an *x* and a *y*, whereas *D* is the subset of *C* whose members also have a *z*. In a grammar-oriented notation, however, what are we to make of a content model named *C*, which allows for children named *x* and *y*, in that order, and a content model named *D*, which allows children *x*, *y*, and *z*, in that order? There is no subset relation here: *C* is the set of elements with exactly two children named *x* and *y*, and no member of *D* is a member of this set. (The XML Schema specification allows the extension of complex types by adding new children at the end, to preserve some grammatically recognizable relation between ancestor and descendant in the type hierarchy, but this has not impressed all observers as satisfactory.)

- Perhaps the worst mismatch between XML and the OO model is the OO idea of packaging data together with the procedures to be used to process it. Since separation of data from processing rules is one of the fundamental tenets of descriptive markup, OO and descriptive markup appear to be fundamentally at odds on this question. On the other hand, many object design patterns stress the use of object composition and observer classes, which effectively allow for some kinds of separation between information and the way it's processed. No general resolution of this problem has been achieved.

XML and the relational model. At first glance, XML and the relational model appear hopelessly at odds. XML data seldom reduces gracefully to tabular representation: elements have variable numbers of children, XML data is typically not even in anything resembling first normal form, and sequence is essential. (It is not enough to get all the words of *Hamlet* out of a database representation of the play; they have to be in the right order.)

Yet, many of the goals of XML and the relational model are similar: application independence, declarative rather than imperative semantics, provision for validation of the data, definition of a generic basic data model that can be used to represent domain-specific concepts and information, with the mapping from the model to real-world semantics firmly under the control of the user. Because so many of the fundamental goals are similar, it appears that although there are many thorny technical issues involved in supporting XML in relational systems, there are fewer philosophical problems than might have been expected.

OPEN QUESTIONS FOR RESEARCH AND DEVELOPMENT

XML is a very young specification, not yet 10 years old. Its surprising uptake presents many opportunities for further work both in practical development of systems and at more theoretical levels. Here are some of the issues left to be resolved.

XML needs better tool support: easier mechanisms for gluing processes together and easier construction of XML pipelines.

Along similar lines, we need better ways of handling multistage processing and better methods for describing the sequence of related schemas through which the data passes.

Users of conventional programming languages desperately need better data-binding tools for marshaling and de-marshaling XML. Here, some problems may be laid at the door of today's tools, which sometimes offer less-

than-stellar support for important constructs. But some problems are related to the mismatch between programming-language data structures and XML.

Some observers think the correct answer is not to improve the impedance mismatch between XML and conventional programming languages, but to eliminate it outright. We can avoid marshaling and de-marshaling entirely if XML becomes a native data type. SQL systems are moving determinedly in this direction, and some programming languages may follow suit. Some systems that support XML natively already, such as XSLT and XQuery, may quietly acquire more of the accoutrements of full programming languages.

By offering tree structures, instead of just lines of characters or tabular structures, XML dramatically enriches the possibilities for representation of documents and other information. Many kinds of information, documents among them, have prominent hierarchical organization and their representation using XML is dramatically more natural and convenient than using competing notations. But the hard fact is that in many kinds of interesting data, hierarchical structures coexist with other, competing hierarchical structures, or with information that resists any kind of hierarchy.

To take a simple example: A book typically has a hierarchical logical structure of front matter, body, and back matter, with the body being subdivided into chapters, sections, subsections, and so on; but books also have a physical structure of volume, gathering, opening, page, column, line. Whenever paragraphs flow across page boundaries—that is, virtually always—these two hierarchies come into conflict. This topic has been of interest to markup theorists for at least 20 years, and new proposals continue to appear: Concurrent markup hierarchies, colored XML, GODDAG (general ordered-descendant directed acyclic graph) structures, just-in-time trees, LMNL (Layered Markup Annotation Language), and range algebras are just a few of the more interesting recent proposals.

XML has inherited from formal language theory as defined by Noam Chomsky in 1957 the notion that a language is a Boolean set of strings.³ Applied to documents and document grammars, this means that documents are either valid and members of the set or else invalid and not members. In reality, some errors are more severe than others, and our systems would be less rigid and brittle if our notion of validity allowed continuous (“fuzzy”) values instead of forcing a black/white distinction. The rigidity of the distinction is one reason that some XML users prefer not to use document grammars. A more flexible notion of validity would make writing flexible applica-

tions possible without giving in to dirty data.

Given the massive proliferation of schemaless XML vocabularies, the need for tools to support grammar induction is increasing: Given a body of XML data, what grammars can be written that describe the data? There are several more or less widely known efforts in this area, from the attempt to generate a grammar for the *New Oxford English Dictionary* in the late 1980s to the industrially oriented grammar induction of the Fred project at OCLC (Online Computer Library Center).

Schemaless or not, the number of XML vocabularies is exploding and unlikely to shrink anytime soon. Both in the context of data integration projects that provide searching over a federation of data sources, and in the context of a single project working with an evolving document grammar, applications of the data-exchange problem to XML are important. Given two schemas S1 and S2, allow the convenient specification of a mapping from S1 to S2 or find such a mapping automatically. Given that mapping and a query against schema S2, translate the query into terms of schema S1 to allow the data to be filtered without first being materialized in schema S2.

How does XML help solve the semi-structured data problem? XML provides a tool for representing and grappling with the data and recognizing the complexity of its inbuilt structure. ☐

REFERENCES

1. Cameron, R. D. 1999. REX: XML shallow parsing with regular expressions. *Markup Languages: Theory & Practice* 1(3): 61-88.
2. Gonnet, G. H., and Tompa, F. W. 1987. Mind your grammar: a new approach to modeling text. *Proceedings of the 13th International Conference on Very Large Databases*.
3. Chomsky, N. 1957. *Syntactic Structures*. The Hague: Mouton.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

C. M. SPERBERG-MCQUEEN is a member of the technical staff of the World Wide Web Consortium, an international membership organization responsible for developing Web standards. He co-edited the XML 1.0 specification and the *Guidelines* of the Text Encoding Initiative. He holds a Ph.D. in comparative literature from Stanford University, but wandered into computing as a graduate student and never came back out. His Erdős number is 6.

© 2005 ACM 1542-7730/05/1000 \$5.00