From Data Representation to Data Model: Meta-Semantic Issues in the Evolution of SGML

CS-95-17

April 1995

Darrell Raymond
Frank Tompa

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Derick Wood

Department of Computer Science
Hong Kong University of Science and Technology
Kowloon, Hong Kong

This paper was accepted for publication in the July 1995 issue of *Computer Standards and Interfaces*.

INTRODUCTION

The ISO SGML standard for document markup has had a tremendous impact on the electronic document community's understanding of its data (ISO SGML 86). The notion of a public, common representation for text suggests a level of data sharing and systems interoperability that was previously unknown. By focusing attention on the structure of a document rather than its appearance, SGML popularized a new approach to document management, one that treats documents as databases, rather than artifacts whose sole function is to be displayed.

By shifting documents from display-centered media to database data, however, SGML has raised new problems that the standard itself was not intended to solve. These problems arise because a printed document is fundamentally different from a database. A printed document is essentially a closed world with a single product. A database, on the other hand, is an open world, with many potential products, and most importantly, with the possibility of change.

Systems that manage change have several responsibilities. First, they must define the operations that cause change, and specify the logical properties of those operations. Second, they must support the definition of constraints that identify valid changes. Third, they must provide techniques for handling concurrent change—that is, change by more than one person at the same time. Fourth, they must preserve the continuity of the data's essence while it changes. The ability to handle these responsibilities constitutes the bulk of a database system's capability to express *semantics*, or the connection between data and the real world.

SGML provides very little functionality for managing change. SGML's basic contribution is the notion of a *document type definition*: a formal, external description of a class of documents. A DTD provides descriptions of static data, and so appears to be similar to a database schema. But where databases treat a schema as part of a complete system for maintaining semantics, SGML explicitly declaims semantics (Goldfarb 90). It is perhaps not surprising, then, that SGML also has little to say about managing change.

There has been much interest in the possibility of providing semantics for SGML and SGML-like systems (Sperberg-McQueen 92). Some of these efforts include mechanisms for representing application semantics (Macleod *et al.* 92) (Sacks-Davis *et al.* 94), standards for hypermedia structures (ISO

HyTime 92), and various systems for transforming structured documents (ISO DSSSL 94). Most of these efforts are much more concerned with querying than with update. The *Guidelines for Electronic Text Encoding and Interchange*, for example, allot about 1% of its 1200 pages to the subject of modification (TEI 94).

In this paper we neither appraise the semantics of previous proposals, nor introduce our own. Instead, we discuss some issues that are 'meta-semantic': issues that lie behind every semantics, but that are peculiar to none. We discuss three issues—equivalence, redundancy, and operations—that are implicit in current systems and approaches. By making these issues explicit, we hope to provide the beginnings of a framework for comparison of the semantics of existing systems, and the development of more advanced ones.

Before considering meta-semantic issues, it will be useful to briefly recount the differing experiences of the document and data processing communities.

DATA MODEL OR DATA REPRESENTATION?

There is a long history of computer processing of both text and traditional data. The communities interested in each type of data are largely disjoint; traditional data has been the province of businesses, and textual data has been the focus of the scholarly and typesetting communities.[1] Both groups put their data online very early, and both suffered from the problem of having that data trapped in proprietary and monolithic systems. What is interesting is that the communities chose different ways to solve this problem.

The document community's dissatisfaction with its early systems led it to focus on two characteristics of the embedded markup used to structure the data (Coombs *et al.* 87). The first characteristic was that in many systems, markup was hidden or otherwise unavailable to the writer. Some word processors, for example, used unprintable character codes to represent markup. Most systems had baroque markup interactions, often resulting from the limited properties of the macro processors that interpreted the markup. Programming these macro processors was an arcane art, and such programs were usually poorly documented and maintained.

1 Even business document production has been generally independent of the database and data processing departments.

The second characteristic of existing systems was that markup was almost always employed to code presentational aspects of documents, rather than structural aspects. Presentational markup was optimized for a specific appearance and not for other purposes. As a result of these two problems, documents tended to become chained to their display system, simply because it was the only environment that guaranteed consistent behaviour.

The solution chosen by the document community was to generalize its markup: to make it accessible to the writer and to encourage the markup of more general properties of texts. The ISO SGML standard popularized the following notions:

1. Markup should be standard and vendor-independent.

2. Markup should be visible and accessible.

3. Markup should be structural, rather than presentational.

The appropriate use of SGML makes significant progress towards achieving these goals (with some minor complications due to markup minimization and other less-commonly employed features of SGML).

The traditional database community's dissatisfaction with its early systems was due to unwanted dependencies embedded in applications (Date 1990). Applications often relied on detailed information about the physical layout of data records; in some cases, even to the extent of knowing the number of bits used to represent data fields. With this kind of knowledge exploited by applications, it was impossible to evolve the database, or even to move it to a new architecture, without rewriting all affected applications. Even the problem of determining *which* applications relied on a given fact about data layout was difficult.

Unwanted dependencies were also present in the use of machine pointers to represent relationships. Pointers provided fast access, but they also embedded certain forms of data access in the system, and thus made it difficult to change the system to accommodate new requirements or to take advantage of new data structures.

The solution developed by the database community was to generalize the semantics of data, by developing data models that described the logical properties of data, independently of how it was stored. The relational model, proposed by Codd, popularized the following notions:

1. A standard data semantics, both static and operational.

2. No hidden access paths to data.

3. No special knowledge of data representation.

Appropriate use of the relational model (or any model with sufficiently well-defined semantics) makes the evolution of a database much easier than does an approach based on a specific data representation.

The document community and database community each suffered from systems that did not provide a clean separation of implementation from application. The two communities arrived at different solutions to this problem, because they had different ideas about what information their systems should preserve.

For the document community, the factor of most permanence was the document. Documents often lasted longer than the systems used to print them. This was particularly true for scholarly documents. While it was important to put documents online, it was just as important to avoid extensive processing each time a new use was contemplated, or each time there was a machine or operating system upgrade. Settling on a common representation for documents was a natural way to avoid these problems, since if data was transportable between systems, then one would have a clear route to leave any given system. The document community thus chose to standardize the representation of data.

For the database community, the factor of most permanence was the semantics of applications. In traditional database applications, data changed so quickly that any given datum was essentially ephemeral. Applications, on the other hand, evolved only slowly. The actual values in bank accounts are continually changing; what is constant is the requirement that the accounts balance—in other words, the maintenance of consistent semantics of the bank's transactions. Focusing on data models was a natural way to ensure these semantics, since data models dealt with semantics directly, and without taking into account the machine, operating system, or data representation. Thus, the database community chose to standardize the semantics of data.

Data interchange was (until recently) largely irrelevant to the database community, since the notion of transmitting (for example) bank account information elsewhere was simply considered too difficult. Too large a volume of data would have to be transmitted, and that data was often changing quickly; in addition, there were problems of privacy and security. Only recently have electronic data interchange standards become important for traditional data.

Data semantics was not irrelevant to the document community, but the definition of semantics did seem to be a difficult problem. Any attempt to standardize semantics, such as that found in the ODA standard, was likely to be criticized as being too limiting or too application-specific (ISO ODA 89). Attempts to define semantics in the scholarly community, most notably the Text Encoding Initiative, similarly met with resistance. Thus, the route proposed by SGML was a reasonable one: promote the notion of application and machine independence, and provide a base on which semantics could eventually be developed, but avoid actually specifying a semantics.

The different solutions found by each community led to a difference in their use of the word ‘data model’. For the database community, ‘data model’ means a common language for describing constraints on data and the effect of operations on that data (Kerschberg *et al.* 76). For the document community, ‘data model’ means a common language in which to express the structure of data. To a database practitioner, SGML is not a data model, since it defines no operators. To SGML practitioners, SGML is a data model for *any* kind of data, since it is possible to express any data structure as an SGML-conformant stream of markup and data.[2]

If documents are never updated, then a document interchange standard may be a sufficient basis for a document database. But the very act of putting documents online changes our concept of documents, and indeed challenges existing ideas about what should be kept fixed and what changeable (Levy 94). Document data may never need to be updated as rapidly and fluidly as financial or other traditional business data, but the need for any update at all carries with it questions about consistency, redundancy, correctness, and effectiveness. These are the issues that are not addressed by a document interchange standard, and should be addressed by any document semantics.

2 This is true but unexciting. It is also possible to represent an SGML-conformant data stream as a directed graph stored in a binary relation, and it is possible to represent anything that we can compute as a stream of 0’s and 1’s on a Turing machine tape. What matters is not possibility, but effectiveness.

NOTIONS OF EQUIVALENCE

Perhaps *the* fundamental semantic issue is equivalence—when are two things the same? Equivalence is important in many contexts:

- If two different forms of data are equivalent, then we may choose to store the one that uses less space
- If two different data items describe the same object, then we should try to ensure that they are consistent
- If two different queries produce the same result, we can query the database with the more efficient one
- If two sets of updates produce the same result, we can modify the database with the more efficient one

A sophisticated notion of equivalence partitions representations into equivalence classes that have the same semantics. SGML's concept of tag minimization, for example, implicitly defines an equivalence class of representations that includes fully expanded markup, fully minimized markup, and variants in between. The virtue of having a variety of representations is that one can choose between them to optimize different tasks. Tag minimization, for example, is designed to make input more efficient, while tag expansion is designed to simplify writing processing routines.

In traditional databases, the equivalence of data is part of the definition of data's domain or type. A relational domain of type `integer` has certain rules of equivalence that are independent of whether it is stored as binary coded decimal, machine integers, text strings, or even a mixture of these. The traditional database requires and expects that (hex) 000a, 'ten', and 0000000000001010 are the same number, and thus is indifferent to which form the number actually takes. Most database models assume that data equivalence is defined, and then build on that to define operational equivalence. In document databases, we are still in the process of defining the domain (the domain of 'text'); thus, the issue of data equivalence is an immediate problem.

The main components of an SGML document are a marked-up text and a document type definition, or DTD. A notion of equivalence for such documents may require the equivalence of DTDs, the equivalence of marked-up texts, or both. In each case, there is a hierarchy of possibilities for equivalence. Some of the levels in the hierarchy are:

- Identity
- Isomorphism
- Structural equivalence
- Equality-preserving operations

The simplest and most direct notion of equivalence is *identity*. Two documents are equivalent under identity if they contain exactly the same characters in the same order, and two DTDs are equivalent under identity if they contain exactly the same content models and identifiers. This type of equivalence is easy to test for, and useful in some contexts (for example, in transmitting documents), but is too strict for many purposes.

Identity may be defined in a slightly less restrictive fashion. Documents that differ only in their use of white space, for example, are typically considered 'identical'. The difference between DOS and UNIX record separators is a sufficiently minor transformation to be performed automatically. Similarly, the substitution of tabs for blanks in most documents should be benign (if the use of tabs was actually indicative of some structure, then by proper SGML usage, this structure should have been captured as an element).

A looser notion of equivalence is *isomorphism*. Isomorphism is identity up to relabelling; that is, all structure is identical except for labels, and there is a one-to-one mapping between sets of labels. Two DTDs can be said to be isomorphic if they are identical except for the choice of generic identifiers. The Text Encoding Initiative uses parameter entities to anticipate the need for this type of isomorphism (TEI 94). Isomorphism is also involved when transforming a DTD with long identifiers into one that satisfies an eight character limit. SGML's notion of variant tag syntaxes is another kind of isomorphic equivalence. Isomorphism in content may take various forms: any one-to-one character transliteration, and some uses of entity references to abbreviate texts, can satisfy the conditions for isomorphism.

At the next level, non-isomorphic DTDs or text may be structurally equivalent. There are many potential forms of structural equivalence for DTDs. We could speak of two DTDs having *element equivalence* if there is a one-to-one map between elements whose content models are equivalent (but not necessarily identical). For example, `x` and `y` are element-equivalent:

```
<!ELEMENT x - - (a|b?)>
<!ELEMENT y - - (a|b)?>
```

More complex versions of equivalent content models are possible. DTDs with element equivalence describe the same class of document structures, but do so in different ways. Another type of structural equivalence for DTDs is *bisimulation*; two DTDs are bisimulation-equivalent if we can find a map (not necessarily one-to-one) between elements, such that mapped elements have equivalent content models.

Structural equivalence can also be defined across marked-up texts. We will say that two documents have *empty-tag identity* if, when we discard the generic identifiers and attributes of all the tags (preserving only their role as start- or end-tag delimiters), then the two documents are identical. Because SGML requires nested markup, such a set of 'empty tags' can denote only one abstract document structure, and thus, it defines an equivalence class of documents.

A still more general notion of equivalence allows variations of structure, so long as some underlying semantic constraints are maintained. Such equivalences are defined by *equivalence-preserving operations*. Consider an operation for replacing all instances of a given element `B` in a DTD with the content model for that element. The new DTD accepts all documents accepted by the earlier DTD, if the tags `<B>` and `</B>` are stripped out. In effect, we can think of the earlier document as a refinement of the new document (or the new document as a kind of 'base' document, and the earlier document as a 'specialization' of the base document). A related notion of equivalence is involved in the CONCUR feature of SGML (where it is possible to ignore a given set of tags in one instance of processing, and ignore the concurring set in another).

Equivalence-preserving operations may produce contraction or expansion. An example of contraction is removing unreachable elements in a DTD; that is, elements that could participate in no document, because they are not part of any derivable sequence. An example of expansion is the introduction of new elements in order to produce certain kinds of normal forms (akin to the process of generating Chomsky normal form for context-free grammars). Equivalence-preserving operations may also, in certain circumstances, convert one kind of SGML structure into another. An exam-

```
<!-- list types as attributes -->
<!ELEMENT list - - (item+) >
<!ATTLIST list
          type     (ordered, bullets, simple) >


<!-- list types as distinct elements -->
<!ELEMENT (ol, bl, sl) - - (item+) >
```

Figure 1: Equivalence of attributes and elements.

ple is the common practice of treating some problems with attributes and others with elements. [3] The type of a list element, for example, may be stored as an attribute on the element, or by having several different kinds of lists, as in Figure 1. Note that this example also employs a second type of equivalence, the isomorphism of generic identifiers.

Equivalence-preserving operations may be specialized to apply only to certain types of data. Simple two-dimensional tables, for example, can be represented as marked-up text, but we tend to require additional semantics for tables, beyond what we would normally expect for running text. Tables can be transposed, for example, while running text usually cannot; thus, transposition should only be permitted for tables.[4] Since transposition of tables is equivalence-preserving, the system should be indifferent to the form (standard or transposed) in which a table is actually represented. Similarly, if it is the case that rows or columns of the table can be interchanged without affecting the table's semantics, then row or column interchange becomes an equivalence-preserving operation (Wang and Wood 93).

To this point, we have only described equivalences between DTDs or between marked-up texts. A fully elaborated notion of equivalence would consider other features of SGML, including entities and attributes. The notion of equivalence should also take into account currently unused possibilities for information representation. ID/IDREF attributes, for example, can be used to represent hierarchies, by having every element point to its parent

3 Our thanks to C.M. Sperberg-McQueen for this point.

4 Or for lists, which are a degenerate form of tables. Note that in the list example in Figure 1, both representations are transposable.

element. Given such a hierarchy, should it be considered equivalent to the conventional one, described by a collection of content models?

We do not insist on any one of these notions of equivalence, or even that there should be only one: we only argue that *some* notion of equivalence is fundamental to a semantics for SGML, and that making this notion explicit and precise is an important step in defining a semantics. The lack of a precise notion of equivalence is directly evident in some SGML problems and in the evolution of related standards. One simple example is the debate about whether documents should be kept in a 'normalized' (unminimized) form, or exactly as they were entered. If a DTD permits tag minimization, there should be no formal objection to storing the unminimized form, the completely minimized form, or any form in between. Users, however, are sometimes confused if the documents they enter are returned to them in a modified form. This problem arises because the user's notion of equivalence differs from that of the system's.

The lack of a precise notion of equivalence is also directly responsible for HyTime's architectural forms (ISO HyTime 92) and TEI's class extension and renaming mechanisms (TEI 94). Roughly speaking, an architectural form is an abstract element definition that stands for a class of actual element definitions that have a similar form—that is, an 'equivalent' content model. This type of element isomorphism was needed because SGML defines none.[5] HyTime chose this kind of technique partly in order to minimize the number of changes one would make to an SGML document; that constraint was itself driven by the fact that SGML has no way to specify or control change.

REDUNDANCY CONTROL

Collections of data are usually interpreted as statements of fact about some world. Most such collections are not minimal; they have data elements that overlap in their statement of fact. When we have two data elements that express the same fact, we have redundancy.

5 There are several additional complexities. HyTime uses attribute values to express isomorphism, partly because of an inability to talk about equivalence between generic identifiers. HyTime also only requires that a given instance of a document conform to the architectural form, not that the whole class of fragments that satisfy the element definition be equivalent to the class that satisfies the architectural form.

It is important to control redundancy if there is a possibility of data being updated. Changing only one copy of a fact leads to inconsistency, and the possibility of contradictory results. Redundant data is also larger than it needs to be, and so increases the cost of storage, indexing, searching, backup, and many other kinds of processing.[6]

Redundancy cannot be controlled unless it is recognized. Recognizing redundancy mean recognizing sameness, and therefore it depends on a notion of equivalence. Conversely, a notion of equivalence brings with it the possibility of sameness, and therefore of redundancy.

The most fully developed notion of redundancy control is found in the work done on normalization of relational schemas (Dutka and Hanson 89). Given a set of dependencies between different parts of the database, we can determine whether a particular database structure will lead to redundant data. Figure 2 shows an example of how to use dependencies to manage redundancy. In 2 (a), we see a simple schema for a database of books stored in various warehouses, and a set of *functional dependencies*. The dependencies tell us that a book with a given ISBN has only one title, and that there is only one quantity of any given book in a single warehouse. With these dependencies and the schema in 2 (a), there is the possibility of redundancy; if a book is stored in several warehouses, there will be several corresponding rows in the table, and each row will have a (redundant) copy of the book's title. This is because the title is only dependent on the ISBN number, and not on the warehouse or the quantity of the book. The schema in 2 (b) removes the redundancy in titles by substituting two relation schemas for the one in 2 (a). With two schemas, each book title will be represented only once, no matter how many warehouses store copies of the books. The process of generating new schemas to remove update anomalies is called normalization. A *normal form* is a schema that is guaranteed to be free of a particular set of redundancies and other update anomalies.

Controlling redundancy does not necessarily mean eliminating it, or always attaining the highest possible normal form. Sometimes it is better to keep, or even add, redundancy for improved performance, especially in distributed situations (Kirkwood 92). What is important is ensuring that 'natural' updates are possible, that redundant data is updated consistently, and

6 In database parlance, removing redundancy and other update anomalies is the process of normalization, and it involves finding a minimal statement of the data. This should not be confused with SGML parlance, where a 'normalized' document is an unminimized one, one with maximal redundancy of embedded markup.

ISBN	title	warehouse	quantity
12345	Data Models	Paris	10
12345	Data Models	London	5
6789	Modelling Text	London	5
6789	Modelling Text	New York	15

$$\text{ISBN} \rightarrow \text{title}$$
$$\text{ISBN, warehouse} \rightarrow \text{quantity}$$

(a)

ISBN	title
12345	Data Models
6789	Modelling Text

ISBN	warehouse	quantity
12345	Paris	10
12345	London	5
6789	London	5
6789	New York	15

(b)

Figure 2: Removing redundancy in a relational schema.

that the form of the stored data does not lead to gratuitous inefficiencies.

SGML's main construct for removing redundancy is entities. If redundant data occurs in a document, a single copy can be kept as an entity, and entity references can be used in place of the redundant copies. One common use of entities is for 'boilerplate' text, such as corporate logos or other common text fragments.

Entities are a technique for removing redundancy, but they are not a technique for managing it. Entities may be defined and used, but they do not guarantee that there are no redundant copies of data in the document. Redundancy control requires more than just a method for removing redundancy; it requires an independent specification of data dependency, that can be used as a standard by which we check whether a given document has any redundancies. Data dependency, in turn, requires a notion of equivalence.

The functional dependencies of the relational model use a value-based notion of equivalence. Given a particular ISBN value in Figure 2, there can be only one title value associated with it. In general, SGML is not used in a way that promotes value-based dependencies. An element that appears more than once in a document, even if it has identical content, is considered to be different simply by virtue of appearing more than once. Words in the text, for example, are values that occur very redundantly. The document community almost never removes this redundancy in order to maintain consistency; instead, it makes use of text editors that apply redundant updates, via string substitution.[7]

SGML has other mechanisms that affect redundancy. Although the use of ID/IDREF attributes is typically thought to be a technique for non-hierarchical linking of data, it is also sometimes used to avoid redundancy. If the reason for pointing to some other part of the text is to avoid copying that data (and perhaps to ensure that updates to the referenced data will be accessible to all referring text), then the ID/IDREF is a means to remove redundancy. This technique is greatly elaborated by HyTime. As with entity references, we do not have an external statement of data dependency, so the use of ID/IDREF to remove redundancy is not a complete solution to redundancy control. Moreover, as with any hypertext system that does not employ typed links, the ID/IDREF facility may express arbitrary non-hierarchical relationships, some that are examples of redundancy removal, and some that are not. Redundancy control typically involves several IDREFs for a given ID, but this is neither necessary nor required.

SGML has several other features whose main purpose is something other than removing redundancy, but that do have an effect on the redundancy of a document. CONCUR, SUBDOC, and marked sections, for example, facilitate the sharing of information. CONCUR allows the sharing of content by two different sets of markup. SUBDOC allows the sharing of an external document that has a different DTD. Marked sections are a method for temporarily escaping certain parts of a document, and thus, in effect, generating a new document that shares all the unmarked sections. Sharing avoids redundancy by reusing a single copy of the data.

Another set of features eliminates information that could otherwise be deduced from the data: examples include RANK, SHORTTAG, and OMITTAG. These features were designed to make the entry of markup less tedious, but they also have the effect of removing redundancy (in markup), and of introducing (or emphasizing the extant) positional dependencies. By

7 We can imagine non-redundant texts, in which every use of a word was an entity reference to a single instance of the word. This technique would have the advantage of promoting consistent spelling (Raymond and Tompa 92). It would also be easier to construct an inverted index, since the word list has already been extracted.

relying on these dependencies instead of entering redundant data, the likelihood of consistent data is improved.

A third set of features creates default values that are implied if no other information is specified: default attribute values and reference concrete syntax are two examples. Defaults are not normally thought of as redundancy control, but they have a similar effect: they represent a shared value, and hence establish a dependency between the parts of documents that employ them. Updating default values transparently updates all those elements that rely on the default values. This reliance on defaults means a reduction in redundancy, and thus an increase in consistency.

OPERATORS

In traditional database usage, a data model is not just a data structure, but an algebra that includes a set of operators for manipulating that data structure (Brodie *et al.* 1984). The relational model, for example, defines both the structure of relations and the relational algebra to manipulate them. The purpose of operators is to provide well-defined methods for accessing and changing data.

SGML defines no operators. Related standards, such as HyTime and DSSSL, do define operators and operator-like capabilities. Since these two standards are likely to be influential in the choice of operators in any future document management system, we will use them as our primary sources of potential operators for SGML.

We can define operators with object-oriented conventions or algebraic conventions.[8] We will first explore object-oriented conventions. These are sometimes referred to as 'CRUD' approaches, since they involve the definition of Create-Read-Update-Delete operations for every object (Kilov and Ross 94). These types of roles for operators are supported in object-oriented languages: C++ classes, for example, have explicit constructors and destructors, and user-supplied methods perform the read and update of the object. Because of its C heritage, C++ does not make a clear distinction between read and update operators, and operators will sometimes both return a value and change some part of an object. Eiffel, on the other hand, makes a point of separating the two types of operator, so as to obtain *referential transparency*: the universal ability to substitute expressions of equal

8 It is also possible to use calculus-like conventions, but we do not investigate these here.

value.

A ‘create’ operator for documents is an activity that is external to the SGML standard. Though SGML does allow for the possibility that documents might be created with standard text editors, creation is more commonly handled by syntax-directed editors or programs that convert from other formats. Construction is not problematic for the most part, because the DTD provides sufficient information about what is required and permitted in every element of the document. One possible problem area has already been mentioned in the section on redundancy control: how do we ensure that entities are used wherever they are appropriate? A construction function might suggest, whenever a datum that is input is equivalent to some existing entity, that the existing entity be used instead. This can only be a suggestion, because we cannot determine in advance whether the author wishes to keep multiple copies of a data value (as might be done if they are only coincidentally equivalent). Similarly, when entities are defined, the construction function needs to scan the whole document for existing fragments that are equivalent, making the suggestion of replacement for each occurrence.

A ‘read’ operator returns the value of some document element. HyTime and DSSSL contain a vast range of techniques for identifying parts of a document, and not much need be said about them here.

An ‘update’ operator changes the value of some document element. HyTime does not have update operators. DSSSL has, on the other hand, perhaps too many operators for producing new document structures from existing structures, and the user can write arbitrarily complex programs to create new operators. The main problem with supplying many operators is the difficulty of understanding their interactions.

The definition of ‘delete’ operators is complicated by *referential integrity*, the requirement that all references have an existing target. Consider deleting an element that has an ID attribute (or consider simply deleting the ID attribute alone). SGML requires that a document have an ID for every IDREF, but it does not specify how to handle such a (potential) deletion. There are at least three possibilities:

RESTRICT Do not delete the ID element if there is a corresponding IDREF element (alternatively, move the ID attribute to a new object and then delete the existing element)

CASCADE Delete all elements with an IDREF corresponding to the ID, and do so recursively if any of them also have an ID attribute

NULLIFY Remove the IDREF attribute from all elements that refer to the deleted ID

The terminology of RESTRICT, CASCADE, and NULLIFY is taken from the relational model. Any of these possibilities will satisfy the SGML requirement; which of the possibilities is chosen depends upon the particular semantics one wants to support. RESTRICT describes a situation in which linked elements must exist; neither the elements nor their relationship are mutable. NULLIFY describes a situation in which the link is optional; removing the target element does not affect the existence of the source element. CASCADE describes a situation in which the linked elements either exist together or not at all. Thus, removing the target implies removing the source. CASCADE is so named because the removal of the source may cause a cascade of removals, if the source itself contains an ID.[9]

We now turn to looking at some of the issues that arise if we adopt algebraic conventions for operators. Algebraic conventions are an offshoot of the mapping of data models to the mathematical notion of an algebra. Algebraic notions are useful for studying the formal properties of operators; they are not so useful for questions of update, because algebraic mathematics has, at best, weak notions of redundancy or resources.

A key algebraic property of any set of operators is *closure*. An operator has the property of closure if its result is of the same type as its input; for example, the integers are closed under multiplication, but not division. Closure allows the results of operations to be further manipulated by the same set of operators. Closure multiplies the power of operators by facilitating their composition, just as documents multiply the power of the finite alphabet by allowing the composition of symbols instead of the definition of new ones. Thus closure is an essential property of any well-defined set of operators (Date 1995).

Another algebraic property is *generation*. We speak of an operator as being a *generator* if repeated applications of the operator can generate any member of the class of objects on which it operates. Generators give us

9 Complications arise when each element of a document can have different update semantics. If, for example, in a chain of deletions caused by a CASCADE, one or more of elements is marked RESTRICT, this causes the whole chain of deletions to be RESTRICTed. Conversely, a chain of CASCADEs will always end at an element marked NULLIFY.

the ability to 'reach' all members of the data space. Closure allows us to construct only members of a given class of objects; generation allows us to construct all members of that class.

DSSSL's Structured Tree Transformation Process can generate any document hierarchy, because it includes a Turing-complete language that can generate, in theory, any object. DSSSL does provides constraints on the output process; no transformation is acceptable unless it can be validated by the output DTD. This is a semantic constraint on update rather than closure. It is worth noting that it is uncomputable to determine whether an arbitrary transformation will satisfy an arbitrary DTD.[10]

The HyQ query language of HyTime, on the other hand, has closure—the results of a HyTime query are always a sequence of HyTime objects—but not generation, since we cannot produce arbitrary HyTime documents as the result of a HyQ query. For example, HyQ does not have an operator that produces a hierarchy of results, as does the `partitioned-by` operator proposed for grammar-defined databases (Gonnet and Tompa 87).

It is both practically and theoretically useful to find a *basis* for a given set of operators. A basis is a minimal and sufficient subset of operators to which all other operators can be reduced. If we can find a basis, then we need only implement the basis, and all other operators can be implemented using combinations of the basis operators. A basis can also simplify proofs of theorems about the operators.

What would constitute a basis for operators on SGML documents? General tree traversal operations, such as those provided by DSSSL, are a natural possibility. DSSSL does not aim for minimality; it would be interesting to deduce a minimal set of operators from DSSSL's tree transformation package. HyTime's addressing methods are not operators, but it is reasonable to infer the operation of selection from them. It is clear that the set of selection possibilities provided by HyTime is not minimal; thus again, it would be interesting to extract a basis. In either case, it is also worth considering generating a basis for some wider class of operations that might subsume either or both of HyTime and DSSSL, rather than trying to force either standard to be an algebra.

Given a set of operators that have the property of closure, we compose the operators into *expressions* that can be applied to objects to produce some desired result. The natural question that arises is: when do two different ex-

10 This observation was first made by Anne Brüggemann-Klein.

pressions generate the same result? We are particularly interested in knowing when two expressions will generate equivalent results, independent of the object they are processing. This particular notion of equivalence is often based on the concept of *identities*. An identity is a rule for rewriting of an expression while preserving its result as, for example, when multiplying by one or adding zero to a number will preserve the number. Given a set of identities, we can develop a variety of equivalent operator expressions and then choose between them based on some metric of efficiency. DSSSL and HyTime do not define identities, though some may be inferred.

Neither the algebraic convention nor the object-oriented convention deal with an issue that will become more important in future document systems: the problem of concurrent update. How should a document behave when more than one user is operating on it at the same time? Algebraic and object-oriented approaches appear to be based on single-user notions of update. Concurrent activities can be reduced to sequences of single-user activities, through locking protocols, but these introduce new problems, such as the possibility of deadlock, and the difficulty of ensuring sufficient concurrency. It may be that we will eventually need a definition of documents that includes a notion of versioning or parallel alternatives to capture the semantics of concurrent update.

CONCLUSIONS

The SGML standard achieved generality not by proposing a syntax for document representation, but by proposing a meta-syntax: a definition of potential syntaxes. In this paper, we provide not a document semantics, but instead, a collection of issues that are meta-semantic: issues involved in the definition of any semantics. The DSSSL draft standard and the HyTime ISO standard have given us some examples of how semantics can be achieved for SGML documents. Now it is time to consider metrics for comparing them and evaluating their feasibility. It is not enough to simply provide operators that seem to have useful functionality. A reliable semantics for SGML documents will have explicit notions of equivalence, effective controls on redundancy, and operators with well-defined properties, such as closure, minimality, and identities. We suggest that consideration of these issues form part of the evaluation of current standards, and part of the design of future systems for managing documents as databases.

ACKNOWLEDGEMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada, by the Information Technology and Research Centre of Ontario, and by an IBM Canada Research Fellowship. We would like to thank Anne Brüggemann-Klein, C. Michael Sperberg-McQueen, Daniel Morales-Germán, and Grant Weddell for their comments on an earlier draft of this paper.

Brodie *et al.* **84** M.L. Brodie, J. Mylopoulos and J.W. Schmidt, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages* Springer-Verlag, New York, N.Y. (1984).

Coombs *et al.* **87** J.H. Coombs, A.H. Renear, S.J. de Rose, "Markup Systems and the Future of Scholarly Text Processing", *Communications of the ACM*, **30**(11) p. 933-947 (November 1987).

Date 90 C.J. Date, *Relational Database Writings 1985-1989*, Addison-Wesley, Reading, Massachusetts (1990).

Dutka and Hanson 89 A.F. Dutka and H.H. Hanson, *Fundamentals of Data Normalization*, Addison-Wesley, Reading, Massachusetts (1989).

Goldfarb 90 C.F. Goldfarb, *The SGML Handbook*, Oxford University Press, Oxford, U.K. (1990).

Gonnet and Tompa 87 G.H. Gonnet and F.W. Tompa, "Mind Your Grammar: A New Approach to Modelling Text" *VLDB '87* p. 339-346, Brighton, England (September 1987).

ISO DSSSL 95 *Information technology—Text and office systems—Document Style Semantics and Specification Language (DSSSL)*, ISO DIS 10179.2, International Organization for Standardization (1995).

ISO HyTime 92 *Information Technology—Hypermedia/Time-based Structuring Language (HyTime)*, ISO/IEC 10744, International Organization for Standardization, Geneva, Switzerland (1992).

ISO ODA 89 *Information processing—text and office systems—Office Document Architecture (ODA)*, ISO 8613-2, International Organization for Standardization (1989).

ISO SGML 86 *Information processing—text and office systems—Standard Generalized Markup Language (SGML)*, ISO 8879-1986, International Organization for Standardization (1986).

Kerschberg *et al.* **76** L. Kerschberg, A. Klug, D. Tsichritzis, "A Taxonomy of Data Models", CSRG-70, Computer Systems Research Group, University of Toronto, Toronto, Ontario (May 1976).

Kilov and Ross 94 H. Kilov and J. Ross, *Information Modeling: An Object-Oriented Approach* Prentice-Hall, Inc. Englewood Cliffs, N.J. (1994).

Kirkwood 92 J. Kirkwood, *High Performance Relational Database Design*, Ellis Horwood, New York, N.Y. (1992).

Levy 94 D. Levy, "Fixed or Fluid? Document Stability in the New Media", *Proceedings of ECHT '94* p. 24-31, Edinburgh, Scotland (September 19-23, 1994).

Macleod *et al.* **92** I.A. Macleod, B. Nordin, D.T. Barnard, D. Hamilton, "A Framework for Developing SGML Applications", *EP '92*, Lausanne, Switzerland (April 7-10, 1992).

Raymond and Tompa 92 D.R. Raymond and F.W. Tompa, "Applying Database Dependency Theory to Software Engineering", CS-92-56, Department of Computer Science University of Waterloo, Waterloo, Ontario (December 31, 1992).

Sacks-Davis *et al.* **94** R. Sacks-Davis, T. Arnold-Moore, J. Zobel, 'Database Systems for Structured Documents', *International Symposium on Advanced Database Technologies and Their Integration*, Nara, Japan (1994).

Sperberg-McQueen 92 C. M. Sperberg-McQueen, "Back to the Frontiers and Edges", Closing remarks at *SGML '92* (October 1992).

TEI 94 C.M. Sperberg-McQueen and L. Burnard (*eds.*, *Guidelines for Electronic Text Encoding and Interchange (TEI P3)*, Association for Computing in the Humanities, Association for Computational Linguistics, Association for Literary and Linguistic Computing (April 9, 1994).

Wang and Wood 93 X. Wang and D. Wood "An Abstract Model for Tables" *Proceedings of the 1993 TUG Meeting*, **14**(3) p. 231-237 Birmingham, England (July 26-30, 1993).