Cobweb: Constrained XML for the Web

Timothy D. McKernan
Bharat Jayaraman

Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260-2000

E-Mail: {tdm4,bharat}@cse.buffalo.edu
Phone: (716) 645-3180 x 111
Fax: (716) 645-3464

Abstract

XML is helping to advance the notion of using the web as a large database. XML can help view, store, manipulate, and transfer semi-structured data that exist in files - often webpages. XML is being developed in part because of the recognized need for a less ad hoc method of handling data than HTML allows. In support of the idea that the web is to be treated like a database, we introduce constraints to XML. Constraints allow for more specific data definitions then XML currently supports, and these definitions may span across multiple webpages across the web. We also define more data types, so that declaring constraints can be more easily facilitated.

1 Introduction

The motivation for this work stems from the observation that many websites, especially those of businesses and organizations, contain structured information, e.g., listings and itemized collections of people, places, events, etc. Such websites can be regarded as databases, in that the structure of the information can be described formally. In deciding how to represent this information, we first note that HTML is not a good choice because it is more oriented towards describing the layout, or presentation, of information. A language such as XML is better suited for this purpose since it provides the concept of *document type definitions* (DTDs) to describe the logical structure of a webpage.

The specification of the logical structure of a website by itself is not enough. In many instances, we also need to ensure that the data in the webpages is semantically coherent. For example, suppose a webpage contains a listing of sales figures for different geographic regions and also an additional field for the total sales. In this case, we would like to state a constraint that the total sales figure is the sum of the regional sales figures. For another example, suppose we had a listing of the monthly sales figures for a given region, we might have a separate field for the average monthly sales and a constraint that the value in this field is the average of the values in all other monthly sales figures. Such semantic conditions are akin to the notion of *integrity constraints* in databases.

While XML has constructs to describe the logical structure of a webpage, one cannot specify integrity constraints using XML. In order to overcome this limitation, in this paper we present a constraint-based extension of XML for specifying both logical structure and semantic coherence of a website. Essentially, constraints may be used in a DTD in order to place restrictions on the values of the elements as well as attributes in the DTD. We provide unary (or domain) constraints, binary constraints (including various comparison operations, as well as aggregation constraints (sum, average, etc.). Although not considered in this paper, the built-in constraint predicates can be augmented with user-defined predicates which may then be used in constraint specifications.

The remainder of this paper is organized as follows: Section 2 provides a bried introduction to XML; section 3 describes our constraint-based extensions; sections 4 and 5 present two illustrative examples of constructs: an on-line brochure example and a product comparison example; finally, section 6 gives the current status of our work and future directions.

2 An Introduction to XML [1]

XML is a markup language that makes handling data easier for programmers. It does this by allowing programmers to create their own tags so that the tags may be context-specific. This is unlike HTML, in which tags are predefined and do not give any information about the type of data being stored.

As an example, consider the following code from an HTML page:

```
<HTML>
<BODY>
<P>John Doe
<P>35
<P>male
<P>Edward Grieg
<BODY>
<HTML>
```

Now look at the same data presented in an XML document:

```
<patient>
    <name>John Doe</name>
    <age>35</age>
    <gender>male</gender>
    <physician>Edward Grieg</physician>
</patient>
```

It is not at all clear what the HTML example is representing, and yet, from looking at the XML example, this is very obviously some sort of medical record. Giving context to data is the first advantage that XML has over HTML. Combine this with one of the many well-written XML parsers that are available, and working with data becomes much simpler and quicker than without XML.

An XML document is created by separating data and information about the data into three distinct sections: the data itself (.xml file), element definitions (.dtd file), and style information (.xsl file). An XML file contains the data being presented, as in the example above. A DTD file contains the document type definition of the data, which is a formal description of the tags that may be used. An XSL file is a stylesheet which contains information regarding the physical layout of the data. An XML-aware browser combines the three files to create a single page that it presents to the user. Depending upon the data, not all of these files are necessary. For example, if data is being sent between servers, and will not be presented visually, then the XSL file is not needed.

The DTD is the most important file for our purposes because it is there we added our constraint definitions. A DTD file already contains element definitions, which may contain other tags, character data (referenced as *#PCDATA*), or some combination of the two. Here is the DTD of an element named *<patient>*. It is composed of two more elements, the *<first_name>* and *<last_name>* elements:

```
<?xml encoding="US-ASCII"?>
<!ELEMENT        patient     (first_name, last_name)>
<!ELEMENT        first_name  (#PCDATA)>
<!ELEMENT        last_name   (#PCDATA)>
```

That is a complete DTD file. A possible corresponding XML file would look like this:

```
<?xml version="1.0"?>
<!DOCTYPE patient SYSTEM "patient.dtd">
<patient>
    <first_name>John</first_name>
    <last_name>Doe</last_name>
</patient>
```

The *DOCTYPE* specifies the root element of the document (*patient*) and the location and name of the DTD file: *SYSTEM* refers to a remote file (as opposed to including the DTD within the XML file), and *"patient.dtd"* is the name of that file. The remaining tags resemble HTML tags.

The DTD also defines attributes. An analogous example of an attribute in HTML is found in the *<img>* tag, in which *width* and *height* may be specified. They are described in greater detail in the next section.

2.1 Attributes in XML

Attributes may be used to define data that the application will make use of but should not neccessarily be seen by the user. Properties and characteristics of elements may be described by attributes. It should be mentioned that the data described by attributes can often be described by elements themselves. This overlap in functionality is a common focus of discussion and debate within the XML community.

Attributes are defined in an attribute list within a DTD. Each attribute has a name to identify it. It has a type, which is usually CDATA (character data), although it may be otherwise (see the Xlink section below for an example). Each attribute may also have a default value, which may be fixed (by the DTD author), or may simply be required (with no default value), or it may be implied (in which case use of the attribute is at the discretion of the XML document author). The values that attributes may contain can be further restrained by specifying a list of possible values in the DTD that are the only possible values to be used.

Here is the DTD of the element and a possible attribute list for an HTML image tag (note: this does not follow the w3's HTML 4.0 specification for an *img* tag):

```
<!ELEMENT IMG EMPTY>
<!ATTLIST IMG
    src         CDATA    #REQUIRED -- URI of image to embed --
    height      CDATA    #IMPLIED  -- override height --
    width       CDATA    #IMPLIED  -- override width --
>
```

Notice how the *img* element has the keyword *empty* in it, because the tag itself does not have data to show the user. The *height*, *width*, and *src* attributes are all used internally by the parser.

Here is a corresponding example of how the attributes would be implemented in an XML document:

```
<img src="http://www.w3.org/img.jpg" width="20" height="30">
```

2.2 Namespaces in XML [2]

XML comprises multiple smaller languages, each of which defines a different domain of solutions to common problems in defining and accessing data. A programmer creating an XML-aware application may choose which of these “sublanguages” to support, based upon the types of data and the way it will be handled. Although XML is a large and complex language when viewed as a whole, any given application need only implement support for the parts of XML that it will use. For example, an application that needs to convert existing XML documents into an XML format that is understood by a third-party database will need a parser that implements XSLT (XSL Transformations), which converts documents from one XML format to another. A second application that needs only to parse data from an XML document, perform operations on the data, and then store the data back into the same file will not need to XSLT, but only the standard XML specifications. Namespaces (described here), and Xlink (described in the next subsection) are examples of these sublanguages. Each one is used in the constrained XML examples in this paper.

XML is intended to be used throughout the web. It is likely that name collisions will occur - that is, more than one DTD will define elements or attributes with the same name that are being used within the same XML document. Namespaces resolves this issue by allowing elements and attributes to have a universal name that makes them unique. The mechanism for doing so is roughly analogous to how we specify webpages. While many website directories have a page called “index.html”, our browser does not confuse which “index.html” to ask for, because we specify in which host and which directory the page exists. Namespaces are similar in that they use a universally unique prefix followed by a local part.

In the following example, the attribute *xmlns* followed by the unique identifier *hospital* specifies that the namespace for *hospital* is *http://www.health.org/patient*. Any element within the scope of the *patient* element will be defined according to the *patient*’s DTD, unless the element declares otherwise. The element *name* may now be used without being confused with other DTDs that implement their own *name* element.

```
<patient xmlns:hospital='http://www.health.org/patient'>
    <hospital:name>Edward Grieg</hospital:name>
</patient>
```

2.3 Links in XML [3]

XML provides a powerful set of ways to connect and link different resources used in XML. For example, besides defining the familiar unidirectional HTML links, which allow one link to point to another resource on a separate page, or instead of pointing to sections within the same page, XML also allows one element to point to multiple resources. Xlink is the language that specifies how to use links in XML. It defines its own namespace.

Xlink provides several global attributes that may be used to define a link. The *xlink* attribute declares the Xlink namespace of the given element.

The *type* attribute specifies if the link is of type *simple*, *extended*, *locator*, etc. *Locator* is the only type we use in this document. The *locator* type allows us to specify the location of resources to be used in our current document. Unlike an HTML link, which, when clicked on, replaces the original document with a new one, we may instead use the *locator* type link to simply make use of some of the information with another page without actually replacing our current page.

The *href* attribute is used to specify the location of the resource, and must conform to being a URI (*Uniform Resource Identifier*). This is where the host name and path are listed.

The *role* attribute is used to describe to applications the function of the resource. Based on this attribute, an application may decide how to display the link.

The *title* attribute gives the name of the link as it would appear to users. This is analogous to the blue text that links have on an HTML page.

Combined, a locator link in XML would look something like this:

```
<patient xlink:href="patient001.xml">Edward Grieg</patient>
```

With this corresponding DTD:

```
<!ELEMENT       patient         (#PCDATA)>
<!ATTLIST       patient
    xmlns:xlink     CDATA
        #FIXED "http://www.w3.org/XML/XLink/0.9"
    xlink:type      (locator)       #FIXED
    xlink:href      CDATA           #REQUIRED
    xlink:role      CDATA           #FIXED "datasheet"
    xlink:title     CDATA           #FIXED "Current Medications"
>
```

The *xlink*, *type*, *role*, and *title* attributes have fixed values, so they do not need to appear in the tag itself. Only the *href* attribute, which should be unique to each patient, need appear in the tag.

3 Constraints in XML

Despite XML's strengths in handling data, it has several weaknesses that hinder it from describing many types of structured data. Specifically, XML does not define data types other than character data (*#PCDATA*), and it does not support operations on the data it defines. We use contraints to deal with these problems.

We have added a two new tags to define constraints in XML:

```
<!CONSTRAINT  constrained_element
    (constraint(&&constraint(&&...)))
>
<!CONSTRAINT  constrained_attribute
    (constraint(&&constraint(&&...)))
>
```

The syntax specifies that *constraint(s)* acts upon the *constrained_element* or the *constrained_attribute*. We have decided to define the *CONSTRAINT* element in the DTD file because the DTD file defines all other aspects of the data's meaning. Note that constraints may be placed on both elements and their attributes.

Constraints are often categorized as either unary or binary constraints. A unary constraint is a constraint that acts upon a single variable, such as a domain constraint, while a binary constraint describes two or more variables. For example, a variable x is given with the constraint that it must be even. This is a unary constraint because the constraint acts only on the variable x. A binary constraint might be like this: there are two variables x and y such that $x < y$. In this case, the variables are dependent upon one another, and so they must exist together and the constraint affects both of them.

Within the context of XML, constraints may be used to check the validity of XML documents before publishing, or the constraints may be solved and used to complete the editing of the document. The latter would make creating pages more convenient for a web author, but must be used in the right places. If a complicated constraint is being fulfilled when dynamically creating a web page, then the process may become uncomfortably delayed. Also, the prospect of multiple solutions introduces the issues of trying to add preferences to the type of solution desired, otherwise the document may not be created as intended.

3.1 Unary Constraints

Both unary and binary constraints may be used in XML. Domain constraints are the unary contraints to be added. In a domain constraint the value being constrained must belong to a domain of values. Consider teachers' webpages in which the class average is listed along with students' grades. The average should be checked to see if it is accurate by computing the average from the list of grades. In this example, a <*class_average*> tag may be given the constraint of having the value of the average of all students' grades.

```
<?xml encoding="US-ASCII"?>
<!ELEMENT        class          (course, average, student+)>
<!ELEMENT        course         (#PCDATA)>
<!ELEMENT        average        (#REAL)>
<!ELEMENT        student        (name, grade)>
<!CONSTRAINT     average        (AVERAGE(student.grade))>
```

Unlike standard XML, in which a tag's data has only a character type, constrained XML also uses integer, real, and string types. These types are specified in the *element* declaration as *#INTEGER*, *#REAL* (as in the above example), and *#STRING*.

Introducing data types makes defining the data simpler both from a conceptual and an implementation perspective. The programmer may immediately go beyond thinking about the mundane task of character conversions and instead concentrate on how to act upon the data. Additionally, if a browser already supports constraints in XML, then no additional programming is needed beyond the authoring of the XML page itself.

Besides making the programmer's job easier, data typing also allows more complex data structures to be created. In XML, as in HTML, tags may be nested within tags. This is similar to objects containing other objects in an object-oriented language; some of the implications are the same. One tag may contain a set of tags describing a student (ie, name, age, id number, grades, etc.). Several student tags may be combined with a similar teacher tag to create a class tag, and so on. Adding constraints to these structures allows more precise combinations to be created.

We make constraints available to the programmer through several predicate functions. The functions return a boolean value depending upon whether or not the specified constraint has been met. The use of predicate functions over a more declarative syntax allows a programming device to be used that is familiar to most web authors.

A simple example of predicate functions is the *count(c_element)* function. The function compares the value of the constrained element to the number of occurrences of *c_element*, and returns true if they are equal, and false otherwise. In a DTD, it would look something like this:

```
<?xml encoding="US-ASCII"?>
<!ELEMENT        packing_slip       (address, item+, total_items)>
<!ELEMENT        address            (#PCDATA)>
<!ELEMENT        item               (#INTEGER)>
<!ELEMENT        total_items        (#INTEGER)>
<!CONSTRAINT     total_items        (COUNT(item))>
```

where the value of <*total_items*> must equal the number of <*item*> tags in a document. A packing slip would use this: the slip lists the total number of items in a package, and itemizes each one. Each item is listed as an <*item*> tag (perhaps with the name of the item in a tag nested within the <*item*> tag). The <*total_items*> tag must be an accurate count of these items so that the packers can check that they have filled the package correctly.

Unlike COUNT(), some of the functions have more than one format. The SUM() function can have one or more parameters: SUM(element), and SUM(element_a, element_b, ...). SUM() always figures out the sum of the data values for each element that is passed in to it, and for every instance of that element in the document. A computer parts distributor might have a document containing a list of products ordered by each customer. A table at the bottom of the page lists how many of each product are needed. SUM() would check how many total speakers, modems, etc, have been ordered by adding the values of the data in the <*quantity*> tag of each product.

```
<?xml encoding="US-ASCII"?>
<!ELEMENT        shipping_orders      (customer*, total*)>
<!ELEMENT        customer             (name, address, items*)>
<!ELEMENT        name                 (#PCDATA)>
<!ELEMENT        address              (#PCDATA)>
<!ELEMENT        items
```

```
                    ((modems | monitors | keyboards | drives)*)
>
<!ELEMENT         modems                (quantity)>
<!ELEMENT         monitors              (quantity)>
<!ELEMENT         keyboards             (quantity)>
<!ELEMENT         drives                (quantity)>
<!ELEMENT         quantity              (#INTEGER)>
<!ELEMENT         modem_total           (#INTEGER)>
<!ELEMENT         monitor_total         (#INTEGER)>
<!ELEMENT         keyboard_total        (#INTEGER)>
<!ELEMENT         drive_total           (#INTEGER)>
<!CONSTRAINT      modem_total
                  (SUM(customer.items.modems.quantity))
>
<!CONSTRAINT      monitor_total
                  (SUM(customer.items.monitors.quantity))
>
<!CONSTRAINT      keyboard_total
                  (SUM(customer.items.keyboards.quantity))
>
<!CONSTRAINT      drive_total
                  (SUM(customer.items.drives.quantity))
>
```

A complete list of unary constraints used in constrained XML is included in the appendix.

3.2 Binary Constraints

Binary constraints allow developers to structure data in more powerful ways than with XML alone. A binary constraint affects two or more values, so that comparisons between groups of values is possible. It is the ability to constrain multiple values at a time that makes ordering possible in a document.

3.2.1 Relational Operators

The common relational operators are included in constrained XML: *lt* (less than), *gt* (greater than), *eq* (equal to), *ne* (not equal to), *le* (less than or equal to), *ge* (greater than or equal to). These operators allow simple comparisons between values:

Consider a manager who needs to prepare a report for a company's intranet that lists his department's weekly expenditures. Preparing the report is a task the manager can easily give to one of his workers. Each time he assigns such a task, he must explain how the document should be setup. For example, the department's total expenditures cannot exceed the department's allowance. The manager can reduce the time it takes to verbally describe a document to a worker by specifying constraints in the report's DTD:

```
<!ELEMENT         allowance                 (#INTEGER)>
<!ELEMENT         total_expenditure         (#INTEGER)>
```

```
<!CONSTRAINT     total_expenditure
                 (total_expenditure le allowance)
>
```

This way, any worker that submits inaccurate information will know upon trying to publish the page that he or she must check this particular constraint in order to correctly edit the document.

3.2.2 Ordering

Binary constraints may also structure groups of tags by ordering tags. The ASCENDING() and DESCENDING() constraints are used to put tags in either alphabetical or numerical order. This immediately allows lists of names, lists of products, chronologies, etc, to be specified in a DTD:

```
<!ELEMENT        faculty        (name)>
<!ELEMENT        name           (#PCDATA)>
<!CONSTRAINT     faculty        (ASCENDING(faculty.name)>
```

Any XML that uses this constraint must fulfill it in the following way: the *<faculty>* tags have been constrained so that they must be ordered alphabetically by the *<name>* tags within them.

A list of faculty members usually has more structure to it than just an alphabetical order. Often a department's webpage will first group faculty according to title (ie, full professor, associate professor, lector, etc), and then alphabetize within the ranks. Ordering by title is an interesting problem because the title names are not in alphabetical order. As far as an XML application is concerned, there is no way to order these titles. There are two approaches to this issue that I will explore. Given only the ASCENDING() and DESCENDING() constraints, a separate tag must be created within the *<faculty>* tag and given a numeric rank, so that the new *<rank>* tags may be ordered:

```
<!ELEMENT        faculty     (name, rank)>
<!ELEMENT        name        (#PCDATA)>
<!ELEMENT        rank        (#INTEGER)>
<!CONSTRAINT     faculty     (ASCENDING(faculty.name)>
<!CONSTRAINT     faculty     (ASCENDING(faculty.rank)>
```

We also provide a mechanism for extracting the *rank* data value from attributes instead of from tags:

```
<!ELEMENT        faculty        (name)>
<!ELEMENT        name           (#PCDATA)>
<!ATTLIST        faculty
     rank        CDATA          #REQUIRED
>
<!CONSTRAINT     faculty        (ASCENDING(faculty.name)>
<!CONSTRAINT     faculty        (ASCENDING(faculty:rank)>
```

In this version, *rank*'s value is stored in an attribute of *<faculty>*, which is accessed by putting the attribute's name in parentheses after the tag name.

The second approach is to enumerate the titles themselves, so that the ASCENDING() and DESCENDING() constraints may be used directly on the <*title*> tags. This will most likely entail creating a new DTD tag, separate from the <*!CONSTRAINT*> tag. The resulting DTD (without specifying the enumeration) is a straightfoward ordering:

```
<!ELEMENT       faculty       (name)>
<!ELEMENT       name          (#PCDATA)>
<!ELEMENT       title         (#PCDATA)>
<!CONSTRAINT    faculty       (ASCENDING(faculty.name)>
<!CONSTRAINT    faculty       (ASCENDING(faculty.title)>
```

3.2.3 Link Traversal

Binary constraints also offer a way to check the integrity of a whole website. The arguments to any constraint may cross over to remote files without a need for a separate syntax. A teacher may make a page for a class that includes the class average. This average may be checked by accessing each student's individual page and reading the student's average.

```
<!ELEMENT          class_average  (#INTEGER)>
<!ELEMENT          student        (EMPTY)>
<!ATTLIST          student
    xmlns:xlink    CDATA
        #FIXED "http://www.w3.org/XML/XLink/0.9"
    xlink:type     (locator)      #FIXED
    xlink:href     CDATA          #REQUIRED
>
<!ELEMENT       student_page   (average)>
<!ELEMENT       average        (#INTEGER)>
<!CONSTRAINT    class_average
                (AVERAGE(student:href(student_page.average)))
>
```

In this example, <*student*> is a link that does not contain an <*student_page*> tag, therefore, the application knows to traverse the link and looks for a <*student_page*> tag (the document root), and then looks for the <*average*> tag, which contains the value we want. In order for the constraint-checking application to know what tags are in a student's page, the DTD for a student page must also be declared in the class page. Multiple DTD's are already supported by XML.

Combining link traversal with ordering allows even more complicated structures to be defined. Consider an school's online brochure, which is composed of a main page containing the table of contents that links to each section of the brochure. The order of the sections is important. In addition, each section has a link to the previous and following sections, as well as back to the main page. Constraints may be used to ensure that the links to *previous*, *next*, and *main* are correct for each page. This model would allow the structure of multiple web pages to be checked. Brochures, tuturials, and well-structured literature (such as plays) could all be checked in this way. The online brochure example will be fully explained in a detailed example below.

3.3 Logical Operators and the *Where* Clause in Constrained XML

There are circumstances where the constraints described so far need to be further restricted, relaxed, or combined. We can achieve this additional expressiveness by making use of logical operators and the *where* clause, the same clause used in SQL.

Suppose the teacher in the last section wants to know not just the class average, but also the disparity between those failing and those passing, so he needs an average for those students with grades below 65, and an average for students with grades 65 and above. Each average has slightly different restrictions, but can be solved with the *where* clause:

```
<!CONSTRAINT     class_average
                 (AVERAGE(student:href(student_page.average)))
>
<!CONSTRAINT     failing_average
                 ((AVERAGE(student:(student_page.average)))
                  WHERE (student:href(student_page.average) lt 65))
>
<!CONSTRAINT     passing_average
                 ((AVERAGE(student:href(student_page.average)))
                  WHERE (student:href(student_page.average) ge 65))
>
```

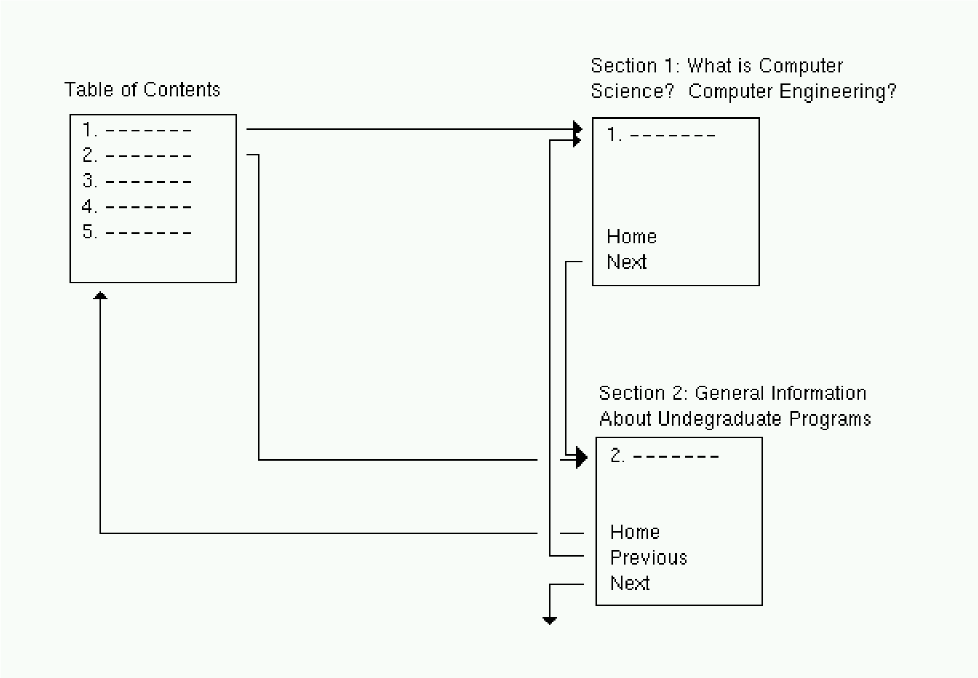
The logical operators used are *AND*, *OR*, and *NOT*. *AND* and *NOT* are included for completeness and to give programmers a familiar way to express these concepts, even though they can already be expressed with the tools described so far. For example, any element that needs two constraints can be expressed with separate constraint tags (as in the ordered faculty examples above) or with the *AND* operator.

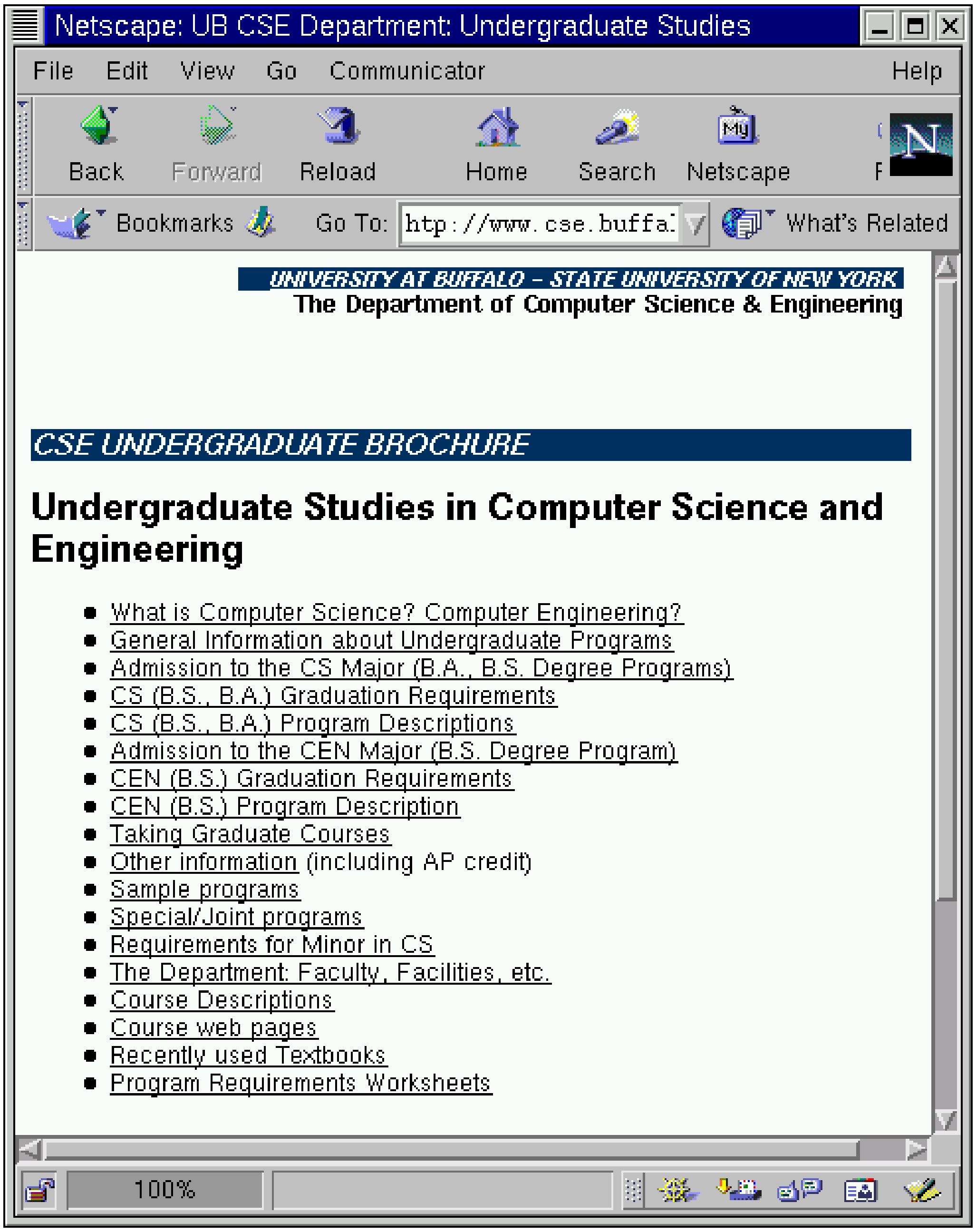
The *OR* operator allows either of two constraints to be fulfilled. An online computer configurator will return information to a server about what type of components to include in a customer's machine. The components must all be carried and supported by the company. If this data is put into an XML then it can easily validated to make sure it has only company-supported hardware.

```
<!ELEMENT        modem        (name)>
<!ELEMENT        name         (#PCDATA)>
<!CONSTRAINT     modem.name
                 (modem.name "lucent" = OR modem.name = "hayes")
>
```

4 The Online Brochure: the first example

Using the concepts described in the earlier sections, we now demonstrate how CobWeb can be used as a solution to a common problem in websites: verifying data content and links across multiple, interconnected webpages. Many websites contain online brochures or other similarly structured webpages, such as tutorials and books. In these webpages a table of contents is defined which links together several ordered sections. The table of contents from the University at Buffalo's Computer Science and Engineering Department undergraduate brochure is given below as an example. The sections refer to each other: Section 1 contains links to the table of contents and to Section 2. Section 2 contains links to the table of contents, Section 1, and Section 3, and so on. The links within sections create two ordered lists, one linking Section 1 through Section 18, and the other linking the sections in the reverse order. Larger and more complex linking schemes are possible. A group of interconnected tutorials, such as the one Sun has for Java technology (http://web2.java.sun.com/docs/books/tutorial/), could have its structure verified by CobWeb. Checking the accuracy of these links is necessary, and yet tedious and impractical for large structures. While this problem cannot be solved using standard XML, CobWeb offers a concise solution.





The DTD is now given. The DTD defines both the table of contents and the sections.

```
<?xml encoding="US-ASCII"?>
<!-- table of contents (toc)
    each listing contains a description and a link to the
    corresponding page
-->
<!ELEMENT      toc                    (toc_section+, toc_loc)>

<!-- the toc_section is a title of a section and a link to
```

```
    that section.  The title is contained within the
    toc_description
-->
<!ELEMENT     toc_section       (toc_description)>
<!ATTLIST     toc_section
              xmlns:xlink        CDATA
                  #FIXED "http://www.w3.org/XML/XLink/0.9"
              xlink:type      (locator)    #FIXED "locator"
              xlink:href         CDATA        #REQUIRED
              xlink:role         CDATA        #IMPLIED
>

<!-- toc_description: both the title of the section, and the
    number of the section, such "1" for "first", etc.
-->
<!ELEMENT     toc_description   (toc_number, toc_title)>
<!ELEMENT     toc_number        (#INTEGER)>
<!ELEMENT     toc_title         (#PCDATA)>

<!-- toc_loc is an empty element, it is not intended to
    display data, but only to store the location of the
    table of contents
-->
<!ELEMENT     toc_loc         EMPTY>
<!ATTLIST     toc_loc
              xmlns:xlink     CDATA
                  #FIXED "http://www.w3.org/XML/XLink/0.9"
              xlink:type      (locator)    #FIXED "locator"
              xlink:href      CDATA        #REQUIRED
              xlink:role      CDATA        #FIXED "toc_loc"
              xlink:title     CDATA        #FIXED "Table of Contents"
>

<!-- checks to see that each section points to the correct
     table of contents -->
<!CONSTRAINT  toc_section
              (toc_section:href(section.toc_loc:href) = /toc_loc:href)
>

<!-- checks that each toc_section points to the correct
     corresponding section -->
<!CONSTRAINT  toc_section
              (toc_section.toc_description.toc_number =
               toc_section->section.toc_description.toc_number)
>
```

```
<!-- checks that the toc_numbers (section numbers) are
     ascending -->
<!CONSTRAINT  toc_section
              (ASCENDING(toc_section.toc_description.toc_number))
>

<!-- section
    contains actual articles, links to toc, prev, and next
-->
<!ELEMENT     section         (toc_description, section_data,
                               toc_loc, prev_loc?, next_loc?)
>
<!ELEMENT     section_data    (#PCDATA)>

<!-- similar to toc_loc - prev_loc is a storage element used
    to hold the location of a previous section
-->
<!ELEMENT     prev_loc        EMPTY>
<!ATTLIST     prev_loc
              xmlns:xlink     CDATA
                      #FIXED "http://www.w3.org/XML/XLink/0.9"
              xlink:type      (locator)    #FIXED "locator"
              xlink:href      CDATA        #REQUIRED
              xlink:role      CDATA        #FIXED "prev_loc"
              xlink:title     CDATA        #REQUIRED
>

<!-- similar to prev_loc - next_loc is a storage element used
    to hold the location of a following section
-->
<!ELEMENT     next_loc        EMPTY>
<!ATTLIST     next_loc
              xmlns:xlink     CDATA
                  #FIXED "http://www.w3.org/XML/XLink/0.9"
              xlink:type      (locator)    #FIXED "locator"
              xlink:href      CDATA        #REQUIRED
              xlink:role      CDATA        #FIXED "next_loc"
              xlink:title     CDATA        #REQUIRED
>


<!-- checks that each prev_loc points to the correct section
-->
<!CONSTRAINT  section.prev_loc
    ((section.prev_loc:href(section.toc_description.toc_number) + 1)
        = section.toc_description.toc_number)
```

```
>

<!-- checks that each next_loc points to the correct section
-->
<!CONSTRAINT  section.next_loc
    ((section.next_loc:href(section.toc_description.toc_number) - 1)
        = section.toc_description.toc_number)
>
```

Now, the XML file that contains the table of contents follows:

```
<?xml version="1.0"?>
<!DOCTYPE patient SYSTEM "brochure.dtd">

<toc>
<!-- toc_loc, a pointer to itself, used to help verify this
    page's location with where each section points to
-->
<toc_loc
    xlink:href="http://www.cse.buffalo.edu/pub/WWW/undergrad/brochure.xml"
/>

<!-- each section of the brochure gets its own "toc_section"
    which contains enough information to present a link to the
    user, the ordering of the section within the brochure, and
    the title of the section
-->
<toc_section
    xlink:href="http://www.cse.buffalo.edu/pub/WWW/undergrad/whatiscs.xml"
>
    <toc_description>
        <toc_number>1</toc_number>
        <toc_title>
            What is Computer Science? Computer Engineering?
        </toc_title>
    </toc_description>
</toc_section>

<toc_section
    xlink:href="http://www.cse.buffalo.edu/pub/WWW/undergrad/general.xml"
>
    <toc_description>
        <toc_number>2</toc_number>
        <toc_title>
            General Information about Undergraduate Programs
        </toc_title>
```

```
    </toc_description>
</toc_section>

<toc_section
    xlink:href="http://www.cse.buffalo.edu/pub/WWW/undergrad/admission.xml"
>
    <toc_description>
        <toc_number>3</toc_number>
        <toc_title>Admission to the CS Major
                   (B.A., B.S. Degree Programs)</toc_title>
    </toc_description>
</toc_section>

<!-- the file continues with the remaining toc_section
    elements that are excluded from this example
-->

</toc>
```

Each section listed within the table of contents has its own XML file. The file for the first section is given below:

```
<?xml version="1.0"?>
<!DOCTYPE patient SYSTEM "brochure.dtd">

<section>
<!-- this information should mimic what was contained in the
    table of contents
-->
    <toc_description>
        <toc_number>1</toc_number>
        <toc_title>
            What is Computer Science? Computer Engineering?
        </toc_title>
    </toc_description>

    <!-- points to the table of contents -->
    <toc_loc
        xlink:href="http://www.cse.buffalo.edu/pub/WWW/undergrad/brochure.xml"
    />

    <!-- since this is the first section, there is no previous
        section to point to, so prev_loc is omitted
    -->

    <!-- points to the next section in the brochure -->
```

```
        <next_loc
            xlink:href="http://www.cse.buffalo.edu/pub/WWW/undergrad/general.xml"
        />

        <!-- contains the actual information to be read in this
            section
        -->
        <section_data>
            The Department of Computer Science (CS) at the State
            University of New York at Buffalo, which was established in
            1967, became the Department of Computer Science and
            Engineering (CSE) in 1998...
        </section_data>
    </section>
```

A *toc* document (Table Of Contents document) is composed of multiple *toc_section*'s, as well as a pointer to itself. Each *toc_section* contains a link to a corresponding webpage with the expected text describing a particular facet of the undergraduate program. Each *toc_section* also contains a *toc_description* which has a number and also a section title.

A *section* element contains a *toc_description* which contains data matching its corresponding *toc_description* in the table of contents webpage. *section_data* is the text of the section. *toc_loc*, *prev_loc*, and *next_loc* are the links to the table of contents page, the previous section, and the next section, respectively.

The correct ordering of each section is checked using multiple constraints. Each *toc_section* is checked to make certain that its corresponding *section* correctly points to the table of contents. Each *toc_section* is also has its *toc_number* compared against the corresponding *section*'s *toc_number* to make certain they are equal. Within each *section* , the *prev_loc* and *next_loc* sections are checked using their *toc_number* numbers. This combination of constraints ensures that the proper structure of the brochure is maintained.

Only four constraints are added, which comprise approximately 25% of the code, yet the effects of the constraints are powerful: without them, a separate and unique application is needed to check this structure.

5 Comparing Products: the second example

In this example, data from other websites is collected, analyzed, and displayed for a user to browse. Unlike the online brochure, in which constraints are used to maintain the structure of the web site, we now use constraints as a way of collecting and modifying data. An application is written that searches popular e-commerce sites (in this case, Barnes & Noble and Amazon.com) for product prices and shipping costs. The results are mapped into an XML page, which adds up the total product plus shipping costs for each site, displays each site's relevant information, and then displays the site with the lowest cost.

This example makes use of two systems - one is the information-gathering application that collects the product information from each site. This application may or may not get information in the correct XML format. It will transform that data into a syntactically-correct XML tree (in memory) and present it to the XML parser. The parser is the second system, where the constraint-checking occurs. It is through the constraint-checking that the parser will add product and shipping costs, and determine the lowest price.

The comparator.dtd is the file that specifies how site and product information must be presented in order for the parser to correctly compare sites. The *comparator* element contains a list of sites as well as a separate *best_buy* element which has the site with the lowest price in it. A site has a name (”Amazon.com”), a list of products, shipping costs, a *total_price* which combines the products and shipping costs, and a link to the store’s website.

The products element may be a book, a cd, or a dvd, each with a title and price, and either a *catalog_number* (for cd’s and dvd’s) or an ISBN number (for books), as well as shipping costs.

Constraints are added to each product: the title, *catalog_number*/ISBN, and price of each product are all derived from each store’s website. The *total_price* element then uses this information about each product to add up the sum of the prices and the shipping costs. Finally, a constraint is placed on the *best_buy* element which says that the *best_buy* element’s *site.total_price* must be equal to the minimum of the *total_prices* of all sites.

The basic comparing mechanism described above can be easily extended to make more complicated analyses. Information regarding when the product will ship (”shipped by” or ”in stock”) could be collected and used to help find the best choice in those cases where the best choice depends upon shipping dates.

The DTD for the comparator looks like this:

```
<!-- add links to site and add constraints -->

<?xml encoding="US-ASCII"?>

<!ELEMENT comparator (site*, best_buy)>

<!ELEMENT site (name, products, shipping, total_price)>
<!ATTLIST site
    xmlns:xlink CDATA     #FIXED "http://www.w3.org/XML/XLink/0.9"
    xlink:type  (locator) #FIXED
    xlink:href  CDATA     #REQUIRED
    xlink:role  CDATA     #FIXED "data location"
    xlink:title CDATA     #FIXED "Click to visit the site"
>

<!ELEMENT name (#PCDATA)>

<!ELEMENT products (book*, cd*, dvd*)>

<!ELEMENT book (title, ISBN, price, shipping?)>
<!ELEMENT cd (title, catalog_number, price, shipping?)>
<!ELEMENT dvd (title, catalog_number, price, shipping?)>

<!ELEMENT title (#PCDATA)>
<!CONSTRAINT title (title = comparator.site:href(product.title))>

<!ELEMENT ISBN (#PCDATA)>
```

```
<!CONSTRAINT ISBN (ISBN = comparator.site:href(product.ISBN))>

<!ELEMENT price (#REAL)>
<!CONSTRAINT price (price = comparator.site:href(product.price))>

<!ELEMENT catalog_number (#PCDATA)>

<!ELEMENT shipping (#REAL)>
<!CONSTRAINT shipping (shipping = comparator.site:href(shipping.ground))>

<!ELEMENT total_price (#REAL)>
<!CONSTRAINT total_price
        (SUM(comparator.site.products.*.price)
         + SUM(comparator.site.products.*.shipping + comparator.site.shipping)
>

<!ELEMENT best_buy (site)>
<!CONSTRAINT best_buy
        (best_buy.site.total_price == MIN(comparator.site.total_price))
>
```

A sample XML file, comparing data from Amazon.com and Barnes & Noble:

```
<comparator>
    <site xlink:href="http://www.amazon.com/">
    <name>Amazon.com</name>
            <products>
                    <book>
                            <title>Fountainhead</title>
                            <ISBN>0451191153</ISBN>
                            <price>7.19</price>
                            <shipping>0.99</shipping>
                    </book>
                    <book>
                            <title>Running Linux</title>
                            <ISBN>156592469X</ISBN>
                            <price>6.99</price>
                            <shipping>0.99</shipping>
                    </book>
            </products>
            <shipping>3.00</shipping>
            <total_price>19.16</total_price>
    </site>

    <site
        xlink:href="http://www.barnesandnoble.com/products/search.jsp"
```

```
    >
            <name>Barnes & Noble</name>
            <products>
                    <book>
                            <title>Fountainhead</title>
                            <ISBN>0451191153</ISBN>
                            <price>7.19</price>
                            <shipping>0.95</shipping>
                    </book>
                    <book>
                            <title>Running Linux</title>
                            <ISBN>156592469X</ISBN>
                            <price>26.96</price>
                            <shipping>0.95</shipping>
                    </book>
            </products>
            <shipping>3.00</shipping>
            <total_price>39.05</total_price>
    </site>

    <best_buy>
            <site xlink:href="http://www.amazon.com/">
                    <name>Amazon.com</name>
                    <products>
                            <book>
                                    <title>Fountainhead</title>
                                    <ISBN>0451191153</ISBN>
                                    <price>7.19</price>
                                    <shipping>0.99</shipping>
                            </book>
                            <book>
                                    <title>Running Linux</title>
                                    <ISBN>156592469X</ISBN>
                                    <price>6.99</price>
                                    <shipping>0.99</shipping>
                            </book>
                    </products>
                    <shipping>3.00</shipping>
                    <total_price>19.16</total_price>
            </site>
    </best_buy>
</comparator>
```

6 Conclusions and Further Work

Our constraint-based extension of XML has several powerful features to add to the language. Using constrained XML allows developers to expand upon the notion of viewing the web as part of a database. Not only is data stored in a specific structure, but it is also checked for validity. Developers can create websites that automatically verify their own structure and dynamically alter and present data without writing separate applications.

CobWeb is now ready to go beyond its current research stage. We plan to implement the constraints described in this paper in order to further demonstrate their usefulness and test the appropriateness of the syntax and design. By taking advantage of XML's modular design we can create a parser that works in conjunction with and extends an existing XML parser. This will allow developers to continue using their own parsers and still take advantage of CobWeb's data typing and constraint features.

References

[1] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. Available at: http://www.w3.org/TR/REC-xml.

[2] World Wide Web Consortium. Namespaces in XML. Available at: http://www.w3.org/TR/REC-xml-names/.

[3] World Wide Web Consortium. XML Linking Language (XLink). Available at: http://www.w3.org/TR/2000/WD-xlink-20000221/

Appendix: A list of constraints in XML

The following is a list of constraints and their definitions that may be added to XML.

common numerical and string operators must be added:

+ addition
− subraction
`*` multiplication
`/` division
= equals
`lt` less than
`gt` greater than
`le` less than or equal to
`ge` greater than or equal to

keywords used for ordering must be added:

```
ASCENDING
DESCENDING
```

`AVERAGE(element)`

the constrained element must be equal to the average of the sum of all occurences of the element passed in.

`CEILING(element)`

the constrained element must be the integer ceiling of the element passed in.

`COUNT(element)`

the constrained element must be equal to the number of occurences of the element that is passed in.

`FLOOR(element)`

the constrained element must be the integer floor of the element passed in.

`ISINT()`

the value of the constrained element must be an integer.

`ISREAL()`

the value of the constrained element must be a real number.

`ISTHIS(element)`

the value of the constrained element must exist as the value of at least one of the values of the occurences of the element passed in.

`MAX(element)`

the value of the constrained element must have the maximum value of all instances of the element in the scope of the element

`MIN(element)`

the value of the constrained element must have the minimum value of all instances of the element in the scope of the element

`SUM(element[, ...])`

the constrained element must be equal to the sum of the elements passed in. Note that this may take several forms. Either multiple elements may be passed in, and their sum taken:

```
SUM(apples.total, oranges.total, bananas.total)
```

or a single element may be passed in, and the application must search for all occurences of the element in the page, and add each instances' value:

```
SUM(product.total)
```