An automatic load/extract scheme for XML documents through object-relational repositories

Frank S.C. Tseng *, Wen-Jong Hwung

Department of Information Management, National Kaohsiung First University of Science and Technology, Kaohsiung 824, Taiwan, ROC

Received 29 May 2001; received in revised form 26 October 2001; accepted 2 January 2002

Abstract

Extensible markup language (XML), a simplified version of standard generalized markup language (SGML), is designed to enable electronic text interchange in the Internet. XML documents have a rigorously described structure that may be analyzed by computers and easily understood by humans. Most current approaches store XML documents in file systems or in relational database systems. However, the nature and the design of file system or relational database schema may cause limitations on fitting with XML document structure. In this paper, we present an automatic load/extract scheme to store and retrieve XML documents through object-relational databases. We propose an architecture, called XML meta-generator (XMG), which, after reading a specific document type definition (DTD), automatically generates the corresponding object-relational database schema (OR-Schema), a DI-Decomposer and a DI-Reconstructor, which are explained as follows:

1. OR-Schema—an object-relational database schema in UniSQL/X format for a specific DTD.
2. DI-Decomposer—a module decomposes XML document instances (DIs) according to the specific DTD format and stores the elements into the corresponding object-relational database.
3. DI-Reconstructor—a module retrieves elements from the object-relational database and reconstructs it to recover the original DI.

These modules make XML documents be automatically decomposed into and reconstructed from object-relational databases in a seamless manner. Moreover, documents stored in the object-relational databases can be managed and inquired more easily than it could be in file systems or relational databases. Useful applications on various documents can also be easily built on top of the target database, such as digital libraries, data warehouses, and data or text mining systems.

© 2002 Elsevier Science Inc. All rights reserved.

Keywords: XML; DI; DTD; Object-relational database

1. Introduction

There are dozens if not hundreds of electronic document formats spread over the Internet. Different software vendors have defined their own document formats. For example, there are doc format of Microsoft Word, Adobe's pdf format, Unix's nroff/troff format (Thompson and Ritchie, 1974), HTML (Lee et al., 1994) and LaTeX format (Lamport, 1986). Although each format has some indispensable merits, they also make users suffer from document interchange in the Internet. Fortunately, thanks to the World Wide Web Consortium (W3C, 2001), a simplified version of standard generalized markup language (SGML; Herwijnen, 1990; ISO, 1986) named extensible markup language (XML; Bray et al., 1997) is established to enable system-independent, platform-independent and complete electronic text interchange environment for authoring and delivery of
information resources across the Internet. XML is a coding scheme that allows various type of information to be conveyed and delivered across the World Wide Web. Its simplicity and user-defined tags make it usable as a semantic-preserving data exchange format. It offers the prospect of a wide range of applications using the Web as a delivery mechanism, from financial transactions, official documents, meeting memoranda, to catalog records, and so forth. Therefore, its applications proliferate rapidly and inspire lots of research works being conducted (Bourret et al., 2000; Shanmugasundaram et al., 1999, 2000).

Basically, a document has content and a logical structure. XML is a scheme designed to capture a document context by separating the document structures and content. That makes it become self-describing and highly flexible for document interchange over the cyberspace. However, an electronic document stored in file systems may suffer from the difficulties of providing interfile consistency, overall privacy, and efficiency when users want to share the data encoded in multiple files. Therefore, to explore the potential of XML documents, and handle heterogeneous file formats in a consistent manner, it is necessary to be able to extract structured data from XML documents and store it in a database, as well as to reconstruct XML documents from data extracted from a database. This can be realized by observing traditional database vendors rush to add DBMSs capabilities to accommodate XML data (Bertino and Catania, 2001). For example, Oracle proposes Internet file system (Oracle, 2001) solution and IBM extends DB2 with XML Extender (IBM, 2001). Some vendors also advocate native XML database management systems, like Tamino (Software AG, 2001) by Software AG and eXcelon (eXcelon, 2001) by Object Design, a division of eXcelon corporation.

Although DBMS vendors are already scrambling to extend their products to manipulate XML, we believe that there is still a need for a platform-independent and lightweight utility to support the loading/extraction of XML documents into/from databases as well, just like Bourret et al. (2000) advocates.

Although XML documents may not be associated with a document type definition (DTD), the work presented in this paper will ignore such cases. Shanmugasundaram et al. (1999) have addressed that the key to make database systems effectively evaluating queries over XML documents is the existence of DTDs (Bosak et al., 1998) or an equivalent, like document content description (DCDs; Bray et al., 2000) or XML Schemas (Microsoft, 2001). Since XML documents in the same type can be categorized into a single DTD, without DTDs or their equivalent, XML will never reach its full potential. Therefore there is the vision of a future in which the vast majority of future files on the web are XML files conforming to DTDs (Shanmugasundaram et al., 1999).

To discuss the relationship between XML and databases, Bourret (2001) distinguishes XML documents into data-centric documents and document-centric documents. The former is designed for structured data for machine consumption, like sales orders, stock quotes, and student records. The latter is supposed to be designed for semi-structured data for human consumption.

Since a DTD represents the structure of a category of documents, however, the nature and the design of file system or relational database schema may cause limitations on fitting with XML document structure. In (Klettke and Meyer, 1999, 2000), it was shown that there are some DTD constructs that can be mapped onto object-relational database structures or not mapped at all. Actually, these mappings work well for applications using XML to represent structured data (i.e., data-centric documents) but not for characteristics of semi-structured data (i.e., document-centric documents). That is why Klettke and Meyer (2000) propose a “hybrid” database approach to represent regular and irregular information in one database and advocate to use database functionality for the most often queried terms and use content-based retrieval for the semi-structured parts. For another comprehensive study about managing complex documents by XML, gentle readers are referred to (Ciancarini et al., 1999).

In this paper, we will focus our work on the applications concerning structured data (i.e., data-centric XML documents, which are characterized by fairly regular structures, fine-grained data, and little or no mixed content), and show that it can be mapped onto an object-relational database schema in a one-to-one correspondence manner, and vice versa. Therefore, under such assumption, theoretically we could develop a tool to translate an XML DTD into a database schema representation automatically and vice versa. Besides, based on such DTD, another tool could also be developed to decompose the same category of XML document instances (DIs), according their specific DTD, into database records. When data in documents are decomposed and stored in database, it makes versatile applications could be easily deployed. For example, applications doing books ordering data aggregations on a specific author could be easily computed by standard object-relational SQL commands.

2. Related works

XML documents are widely used in Web information systems (WiumLie and Saarela, 1999). For example, Bapst and Vanoirbeek (1998) develop an electronic platform for production of Requests for Proposals. Royappa (1999) implements a common Web-based storefront to a group of merchants for electronic commerce.
Prior works on decomposing SGML/XML documents into database records can be categorized into the following categories:

1. Use relational databases as a repository to store decomposed SGML/XML documents. The works established by Macleod (1990), Schouten (1989), and Shanmugasundaram et al. (1999, 2000) fall into this category. Since relational model uses flat tables to represent various kinds of data relationships, it is not versatile enough to capture all possible structures in a DTD. Even it is possible to model a DTD structure by relational schema, it appears to be cumbersome. To store XML documents in a relational database system, Shanmugasundaram et al. (1999) have pointed out that two complications should be addressed, namely set-valued attributes and recursion. But the traditional relational model does not support set-valued attributes. Therefore, foreign key reference mechanism should be employed to accommodate this dilemma, which may downgrade the query performance.

2. Design of a query language specifically for semi-structured data (XML is tree-structured, with each node in the tree described by a label). For example, Ishikawa et al. (1999) has proposed a query language named XQL, designed by combining SQL and OQL features. Buneman et al. (1996) have presented a query language, together with its optimization techniques, for unstructured data, and Abiteboul et al. (1997) and McHugh et al. (1997) have developed a database management system, named Lore, for semi-structured data. Recently, Deutsch et al. (2001) also propose a query language for XML in W3C.

3. Use object-relational databases as a repository to store decomposed SGML/XML documents. Since object-relational databases use versatile class structures to represent various kinds of data relationships, it is good enough to capture all possible meaning in a DTD. However, prior works, for example Christophides et al. (1994) use hard-coded approaches to develop such tools. That is, tools that are expected to be a generic one should be parameterized according to DTDs. Thus, different version of tools should be developed for different DTDs. Klettke and Meyer (1999) have shown that, by using the partitioned normal form (Roth et al., 1988), data-centric XML documents can be straightforwardly mapped onto object-relational databases to some extent. However, they also pointed out that some document-centric XML structures cannot be easily mapped onto database structure, namely mixed-content and alternatives. Besides, in (Klettke and Meyer, 2000), for those that cannot be easily mapped onto database structure, they claim these structures would be mapped onto database attributes of a user-defined type XML and kept as is. Finally, they conduct a statistical approach to find a type of optimal mapping based on the DTD and statistics.

Although applications employing XML to handle data-centric documents are prolific, the database schemas for storing DIs of different DTDs are always constructed manually in prior works (Ishikawa et al., 1999; Macleod, 1990; Zhang, 1995), which really frustrates the users and limits the proliferations of information sharing.

In the following, we will explore the one-to-one mapping relationships between XML DTDs and object-relational database schemas and propose an automatic load/extract scheme to store and retrieve XML documents through object-relational databases. We have designed a system architecture, called XML metagenerator (XMG), which, after reading a specific DTD, automatically generates the corresponding object-relational database schema (OR-Schema), a DI-Decomposer and a DI-Reconstructor, which are explained as follows:

1. OR-Schema—an object-relational database schema in UniSQL/X (UniSQL, 1996) format for a specific DTD.
2. DI-Decomposer—a module decomposes XML DIs according to the specific DTD format and stores the tag items into the corresponding object-relational database.
3. DI-Reconstructor—a module retrieves tag items from the object-relational database and reconstructs it to recover the original DI.

These modules make XML documents be automatically decomposed into and reconstructed from object-relational databases in a seamless manner. Moreover, documents stored in the object-relational databases can be managed and inquired more easily than it could be in file systems or relational databases. Useful applications on various documents can also be easily built on top of the target database, such as digital libraries (IEEE, 1996), data warehouses (Inmon, 1993; Inmon and Kel- ley, 1994; Kimball, 1996).

More specifically, the contributions of this paper are two folds:

1. We propose a scheme and conduct a solution to build object-relational schemata for data-centric XML documents automatically, which avoid the mistakes and inconsistencies caused by manual analysis of DTDs.
2. By using object-relational database as the repository, the whole semantics of DTDs can be captured more easily. Besides, the internal structure can be linked more efficiently due to the implicit join mechanism of object-relational databases.
When the target databases are all well established, they form a versatile digital library and are the central repositories for document retrievals and document statistical analysis.

Our paper is organized as follows. Section 3 is a basic introduction for XML. In Section 4, we show the automatic load/extract architecture and explain the feasibility. Section 5 discusses how to employ yet-another compiler-compiler (YACC) for implementation and explores the one-to-one correspondence between XML DTDs and object-relational database schemata. Finally, we conclude and outline future works in Section 6.

3. A basic introduction to XML

We briefly introduce XML in this section. The examples used in this section will continuously be employed to illustrate the transformation process of our approach.

The XML standard consists of the following three main parts (Light, 1997):

1. XML-language: The language specification. It specifies the syntax of XML and defines the crucial concepts of valid and well-formed XML documents.
2. XML-link: A set of built-in linking facilities for linking within and between XML documents and other Web resources. It supports for link roles and better control over the behavior of links than HTML provided.
3. XML-style, or XSL: Using the cascading style sheets (CSS) mechanism, XML documents can be freely styled. Besides, based on the document style and semantics specification language standard (DSSSL), XML can be formatted by its own style sheet language extensible stylesheet language (XSL) as an alternative to CSS.

Since the focus of this paper is how to store data items encoded in XML documents into databases, we will mainly concentrate on the first part, namely XML-language, of XML standard.

From the viewpoint of XML-language, a DTD defines the logical structure of the same category of XML DIs. It is like a schema for XML documents. In the following, we briefly introduce the relationship between DTDs and DIs.

3.1. XML document instances

The content of a DI should conform to the structure of its corresponding DTD. (We explain the DTD specification in the following section.) Therefore, an XML parser can be used to verify various DIs by referencing the DTD to check whether they are valid or not. Fig. 1(a) is an example of memorandum and Fig. 1(b) is its XML DI representation.

Fig. 1(a) is only meaningful to humans, but cannot be analyzed by computers. However, Fig. 1(b) is a suitable one for computer processing, since the sentence meanings are fully marked up. Note that "’" is used to replace the apostrophe (’), since it is a reserved character in XML specification.

(a) To: Jeff Lee
    From: Paul Wang

    There is a recital tomorrow evening in Taipei. I will wait for you in front of Hotel Hilton at “6:00 P.M.”

    By the way, don’t forget to bring me the book you have borrowed last weekend.

    See you soon.

(b) <xml version = “1.0”?>
    <MEMO STATUS = “PUBLIC”>
    <TO>Jeff Lee</TO>
    <FROM>Paul Wang</FROM>
    <BODY>
    <P>There is a recital tomorrow evening in Taipei. I will wait for you in front of Hotel Hilton at “6:00 P.M.”</P>
    <P>By the way, don’t forget to bring me the book you have borrowed last weekend.</P>
    </BODY>
    <CLOSE>See you soon.</CLOSE>
    </MEMO>

Fig. 1. (a) A simple memorandum and (b) DI of the memo.
The logical structure of a document consists of a logical sequence. For example, the “TO” part is followed by a “FROM” part, and then a “P” part (which means “paragraph”), followed by an optional “Q” part, and finally ends with a “CLOSE” part. In Fig. 1, any logical structure starts from a start tag `<component>`, and ends with an “end tag” `</component>`, and the element name `component` defines its physical meaning. For instance, the start tag `<MEMO>` and end tag `</MEMO>` indicate that this document is a memorandum. The logical structure of a DI can be represented as a tree structure, and the content embedded in the start tags and end tags corresponds to a node in the tree structure. To define the logical tree structure systematically, we have to define the DTD, which is explained in the following section.

3.2. XML document type definitions

A DTD is represented as a very rigorous formal language in BNF. All values are assumed to be string values, unless the type is ANY. A DTD specifies the structure of an XML element type by specifying the names of its subelement types and attributes by using the following keywords to define the logical structure of a document:

1. ELEMENT—Used to define an element type, which corresponds to an element of a DI conforming to the element type.
2. ATTLIST—Used to define the attributes and attribute values of an ELEMENT.
3. ENTITY—Used to define special characters, parameter substitutions, etc.

For the MEMO example in Fig. 1, the corresponding DTD adopted from (Herwijnen, 1990) is as Fig. 2 illustrates. We explain these keywords in the following sections.

3.2.1. ELEMENTs in DTDs

When an ELEMENT is defined in a DTD, it can be used as an element in any DI, which conforms to the DTD. The declaration of an ELEMENT is of the following form:

```
<!ELEMENT element_type (content)>
```

In DTD declarations, the part beginning with “<!” and ended with “>” is regarded as a “markup declaration”, and all text embedded between “<!-” and “-->” is just used for comment. We call the (content) as a “model group”, which is used to define the constitutions of the element type. For example, a model group (#PCDATA) means the element type is defined to be a character string.

In a model group, it may contain “connectors” and “occurrence indicators”. The former is used to indicate the occurrence sequence of the involved elements. The latter indicates the occurrence frequency of each elements contained in the model group.

In Table 1, we list all possible connectors used in an element type. Two element types connected by the connector “|” indicates that the corresponding elements should occur in the regarded sequence. Element types separated by “?” means that they are alternatives. Only one of their corresponding elements can occur.

In Table 2, we also list all possible occurrence indicators. An element with the element type marked with “?” is supposed to be omitted or to appear exactly once.

1 In SGML specifications, there is another connector “&”, which means all of the element types have to appear but the sequence is arbitrary. However, in XML specifications, this connector is forbidden.
A element should appear once or more than one times, if its element type is marked with ‘?’ An element can be omitted or to appear more than once, if its element type is marked with ‘*’. Besides, for comprehensively understanding, a DTD can be formalized to an abstract tree structure. For example, in Fig. 3, the tree structure shows the logical element types of the MEMO DTD and their nesting. A DI for MEMO contains codes for the element type MEMO, which contains codes for the element types TO, FROM, BODY, and CLOSE. The element type BODY contains P, which in turn contains Q.

### 3.2.2. ATTLIST of ELEMENT

ATTLIST is used to describe the attributes of an ELEMENT. It has the following declaration format:

```xml
<!ATTLIST element_type attribute_name type default>
```

where

1. element_type—denotes which ELEMENT the attribute_name belongs.
2. attribute_name—denotes the attribute name.
3. type—used for declaration of attribute type. There are 10 type settings can be used: CDATA, Enumerated, NMTOKENS, ENTITY, ENTITIES, ID, IDREF, IDREFS, and NOTATION.
4. default—to set the attribute default. The available alternatives are as follows:
   (a) #FIXED: denotes the attribute value is fixed.
   (b) #REQUIRED: denotes it should appear and cannot be omitted.
   (c) #IMPLIED: denotes it is optional, and can be omitted.
   (d) Default string: If there is no attribute value assigned to this attribute, then use the default string as the attribute value.

Attributes can be used to describe the characteristics of an element type. It appears as follows:

```xml
<element_type attribute_name = value>
```

For example, in Fig. 1(b), there is an attribute STATUS with value PUBLIC in the element type MEMO. That denotes the memo is not confidential, and can be opened for public.

### 3.2.3. ENTITY

There are two types of ENTITIES in XML DTDs:

1. Internal entities: They are shorthand for a piece of XML markup that is held within the document. They can be used for standard phrases, expansions of abbreviations, and even for single characters (such as the entity reference ‘apos;’ in Fig. 1(b)).
2. External entities: External entities can be text or binary. An external text entity contains text data considered to form part of the XML document. It allows you to build XML documents in chunks. An external binary entity is basically anything that isn’t regarded as though it is XML-encoded. Each binary entity has to associate with a notation. A notation describes a type of resource, which is just a file type, such as GIF or JPG.

The processing of external entities is not covered in this paper. Gentle readers are referred to (Goldfarb, 1990; Herwijnen, 1990; Light, 1997).

### 4. Automatic load/extract of XML documents in databases

#### 4.1. The goal

The primary objective of our framework is to design a meta-tool, called XMG, which, after reading a specific DTD, automatically generates the corresponding object-relational database schema (OR-Schema), a DI-Decomposer and a DI-Reconstructor.
These modules make XML documents be automatically decomposed into and reconstructed from object-relational databases in a seamless manner. With this meta-tool, we may store any type of XML documents into object-relational databases and develop various applications on top of these databases, such as data warehouse, digital library applications.

4.2. The architecture

In Fig. 4, the whole architecture of our approach is depicted. We explain this architecture in three folds:

1. XML documents: XML documents contain a set of DIs which conform to a finite set of DTDs. For example, we use DI_{x} to denote DIs conform to a specific DTD_{x}. All DTDs are input into the Meta-generator by off-line operations.

2. Off-line operation environment: For each specific DTD, the meta-generator will generate the following three components:

   (a) OR-Schema—an object-relational database schema in UniSQL data definition language (DDL) format for a specific DTD, which will be used to automatically construct the corresponding database for DI-Decomposer to use it as a DI repository. We will show that there is a one-to-one correspondence between any XML DTD and object-relational schema in Section 5.3.

   (b) DI-Decomposer—a module reads XML DIs according to the specific DTD format and decomposes the tags and attributes, then generates INSERT commands of UniSQL data manipulation language to store them into the corresponding object-relational database described above.

   (c) DI-Reconstructor—a module retrieves tag data and attribute values from the object-relational database and reconstructs the original DI according to the original DTD structure for electronic interchange. Since the original DTD structure has an one-to-one correspondence with the object-relational schema, the conversion can be easily implemented.

3. On-line operation environment: When a UniSQL DDL command was generated by XMG, it will be automatically put into the object-relational database management system to construct the database schema. Therefore, the database is ready for on-line operations. Note that XMG will construct an object-relational database, a DI-Decomposer, and a DI-Reconstructor for each DTD. Therefore, all DIs in the same category, which obey the same DTD, will be stored in and retrieved from the same database by the corresponding DI-Decomposer and DI-Reconstructor, respectively. The database now is versatile for various applications, such as workflow management (Royappa, 1999), data warehouse constructions (Inmon, 1993; Kimball, 1996), document digital libraries (IEEE, 1996) and full text search.

5. Mapping rules and implementation

5.1. Employ YACC capabilities

We use YACC (Johnson, 1979) to implement the XMG. The XML DTD specification is built in BNF format in the YACC program. Besides, the mapping between DTDs and object-relational database schema is

![Fig. 4. The automatic load/extract architecture.](image-url)
coded in the action routines in the corresponding DTD specification, which will be generated in YACC format to produce the DI-Decomposer. The whole process is as Fig. 5 illustrates.

Besides, since the DTD is already represented in object-relational database schema, we can derive the DI-Reconstructor directly from the OR-Schema.

5.2. The mapping between XML DTD and YACC grammar

After reading a DTD specification, XMG translates the DTD into a context-free grammar in YACC format. Together with some pre-defined action routines, the output will become the corresponding DI-Decomposer.

The transformation rules are classified into three categories:

1. Mapping rules for connectors.

We explain these categories in the following. Note that we adopt the naming convention of YACC by using lower case identifiers to represent non-terminal symbols and upper case identifiers to denote terminal symbols. Therefore, if an identifier with upper case chars in DTD is a non-terminal symbol, and then all its chars will be translated into lower case.

5.2.1. Mapping rules for connectors

There are two connectors that may occur in an ELEMENT declaration, namely ‘|’ and ‘,’. We explain the mapping rules as follows:

**Rule 1.** For connector “|”, it means alternatives of an ELEMENT declaration. Therefore, the transformation of <!ELEMENT X (Y1|Y2|…|Yn)> is

```
x : X y1 EX
    | X y2 EX
    | …
    | X yn EX
```

Note that since terminal and non-terminal symbols are represented in upper and lower case symbols in YACC, respectively, lower case letters (“x” and “y”) are used to represent the corresponding ELEMENTs. Moreover, the upper case symbols “X” and “EX” are used to match the terminal symbols “<X>” and “</X>” in any DI, respectively.

**Rule 2.** For ELEMENTs connected by the connector “,”, they should occur in the regarded sequence. Therefore, the transformation of <!ELEMENT X (Y1, Y2, …, Yn)> is

```
x : X y1, y2, …, yn EX
```

For instance, consider the MEMO ELEMENT declared as

```
<!ELEMENT MEMO (TO, FROM, BODY, CLOSE)>.
```

Then, the corresponding YACC grammar is

```
memo : MEMO to from body close EMEMO.
```

In this grammar, “to”, “from”, “body”, and “close” must be further transformed according to their corresponding ELEMENT declarations. Besides, the upper case symbols “MEMO” and “EMEMO” are used to match the terminal symbols “<MEMO>” and “</MEMO>”, respectively.

5.2.2. Mapping rules for occurrence indicators

There are three occurrence indicators that may occur in an ELEMENT declaration, namely “?”, “*”, and “+”. We explain the mapping rules as follows:

**Rule 3.** For ELEMENTs attached with occurrence indicator “?”, it means optional occurrence. The transformation is just adding an empty rule into the corresponding ELEMENT grammar.

For example, if we have the DTD declaration as Fig. 2 illustrates. Then, after substituting the ENTITY “%doctype;” by MEMO, we have the declarations of ELEMENTs MEMO and CLOSE in the following:

```
<!ELEMENT MEMO (TO, FROM, BODY, CLOSE?)>
```

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

Then, the corresponding YACC grammar for MEMO is as that illustrated in Section 5.2.1. But the grammar for CLOSE will be added with an empty rule:

```
close : /* empty rule */
    | CLOSE PCDATA ECLOSE
```

**Rule 4.** For ELEMENTs attached with occurrence indicator “*”, it means iterative occurrence of the ELE-
MENT for zero or more times. The transformation can be obtained by replacing the ELEMENT by a non-terminal symbol named “element_star”, where element is the ELEMENT attached with “*” occurrence indicator, and then making the grammar rule of ELEMENT to recursively define itself.

For example, consider the following ELEMENT declaration as appeared in Fig. 2:

```
<!ELEMENT BODY (P*)>
```

Then, the corresponding YACC grammar rules for “BODY” and “P” are as follows:

```
body : BODY p_star EBODY

p_star : /* empty rule */
     | p_star P

```

Note that since the ELEMENT may occur zero times, the non-terminal “p_star” is added with an empty rule.

This transformation rule is also applicable for ELEMENTs declared with occurrence indicator “+”. However, some action routines should be added into the rule to check the occurrence frequency is greater than or equal to one.

5.2.3. Mapping rules for combination of the above

We now further explore the transformation for complex combinations of the connectors and occurrence indicators as follows:

**Rule 5.** For an element type X declared as `<!ELEMENT X (Y1 θ Y2 θ ... θ Yn)σ>` where θ ∈ {‘|’, ‘‘} represents connectors, and σ ∈ {‘?’ , ‘*’ , ‘+’} represents occurrence indicators, the transformation can be obtained by combining Y1, Y2, ..., and Yn and their occurrence indicator σ to declare another non-terminal symbol named “y1_y2_..._yn_σ”, and define this non-terminal symbol by applying all of the rules discussed before. In practical, to make the composed non-terminal symbol syntactically correct, the occurrence indicator σ should be represented by its alphabet name.

For example, in Section 5.2.2, the grammar rule for “P” is not explained. We now further explore Rule 5 by using ELEMENT P as the example. In Fig. 2, ELEMENT P is declared as:

```
<!ELEMENT P (#PCDATA | Q)*>
```

To convert this complex declaration smoothly, we simply combine the alternatives to declare another non-terminal symbol named “pcdata_q_star”, and define this non-terminal symbol by applying all of the rules discussed before:

```
p : P pcdata_q_star EP
```

5.3. The mapping between XML DTD and UniSQL/X database schema

As an object-relational database management system, there are some features in UniSQL/X that can be utilized to support our mapping mechanism. We describe them as follows:

1. User-defined data types—Since XML DTD supports unlimited element type definitions, the user-defined data type in UniSQL/X will be employed to cope with this features.

2. Set-valued type—An element type definition in DTD may contain another element type definitions, or even recursively define itself. In relational databases, we need to associate foreign keys with primary keys in other tables to implement set-valued type. However, since UniSQL/X supports some versatile set-valued data types defined on a domain or a domain list (such as SET, MULTISET, and SEQUENCE), we may map the element type definition into a class definition in a natural way.

3. Encapsulation—The functions to be utilized by a DTD element type can be implemented by the class methods to form an encapsulation.

We have tried to find out all possible combination of a DTD element type definition, and their corresponding mapping patterns in UniSQL/X DDL format. To achieve this goal, we analyze the relationships between ELEMENT and ENTITY and that between ATTLIST and ELEMENT, and conclude the following rules.

**Rule 6 (Mapping rules for an ELEMENT).** This can be distinguished into the following three principles:

(a) For each element type, create a user-defined class with the same name to store it.
(b) Convert “#PCDATA” or “CDATA” into string data type in UniSQL/X.
(c) The “model group” will be mapped to a SEQUENCE (also called as LIST) type in UniSQL/X, and each element type in the model group will be mapped to a user-defined class.

**Rule 7 (Mapping rules for an ATTLIST).** Any attribute name of an ELEMENT E will be mapped to an attribute of the corresponding user-defined class E, and its data type is string. To distinguish such an attribute from
For example, if there is an element type with its ATTLIST declaration in a DTD as shown below:

```
<!ELEMENT E (A, B)>
<!ATTLIST E attribute _name type default>
```

Then this will be converted into the following UniSQL/X schema.

Create class E (  
  E sequence(A, B),  
  ATT_attribute_name string default 'default'  
);  

Based on these rules, we summarize the rules for mapping DTD constructs to UniSQL/X DDL in Table 3, where Element represents the corresponding element type. In our approach, all primitive values are regarded as string data type. Therefore, a datatype in Table 3 is used to denote either string or a complex data type, which needs to be defined further according to the DTD.

According to the rules in Table 3, the MEMO DTD example shown in Fig. 2 can be mapped to the schema shown in Fig. 6. In Fig. 7, we also show the tree structure representation of the derived schema.

```
INSERT into TO (TO) values (“Jeff Lee”) to :TO
INSERT into FROM (FROM) values (“Paul Wang”) to :FROM
INSERT into BODY (BODY) values ((INSERT into P(P) values ("There is a recital …", INSERT into Q(Q) values("6:00 P.M.")]),
  INSERT into P(P)
  values ("By the way, …")) to :BODY
INSERT into CLOSE (CLOSEE) values (“See you soon.”) to :CLOSE
INSERT into MEMO (MEMO, ATT_status) values
  ([:TO, :FROM, :BODY, :CLOSEE], “public”)
```

Fig. 8. INSERT commands derived by DI-Decomposer for MEMO class.
It is obviously that Figs. 7 and 3 are exactly the same structure, which means they have one-to-one correspondence on the nodes. Therefore, based on this mapping scheme, after reading the MEMO DI shown in Fig. 1(b), the DI-Decomposer will produce some INSERT commands of UniSQL/X as Fig. 8 depicts to decompose the data into the class MEMO. Finally, the data inserted into the class MEMO are shown in Fig. 9.

6. Summary and future directions

In this paper, we have elaborated an automatic load/extract scheme for storing XML documents into object-relational databases. The mapping between XML DTDs and object-relational database schema is shown to be one-to-one correspondence. Therefore, the whole process can be smoothly implemented in a seamless manner. A more complicated prototype focus on the automatic load/extract scheme for SGML documents has been implemented to prove the feasibility.

Besides, for various applications of XML used to format official documents, we have defined DTDs for clinical records and convert some sample records into XML documents and stored them in object-relational databases for versatile applications. The target database is extremely useful, since we may run search engine on top of it to search documents more efficient, apply data mining techniques to find some valuable information, and so forth.

The contribution of our work can be summarized as follows:

1. We propose a simple and symmetric mapping scheme between DTDs and object-relational schemata—The mapping is shown to be one-to-one correspondence, and the whole process is automatic without manual operations or design. This avoids the mistakes and inconsistencies caused by manual analysis of DTDs.

Besides, since all produced modules are executed on database server side, all users in such a system need not to care all details about the execution process.

2. Semantics and Efficiency—By using object-relational database as the repository, the whole semantics of DTDs can be fully captured and queries posed on the object-relational databases is supposed to be more efficient than relational databases do. This is because object-relational database management systems use implicit join operations on retrieving XML documents, but relational database systems must adopt explicit join operations to accomplish the same task.

In the next step, we intend to enhance the prototype to manipulate XML documents for Web applications in a more complete and subtle way. This work will be combined with the style sheet languages CSS or XSL to format XML documents when reconstructed from databases.

Moreover, since DTDs do not offer sufficient information for the data types and size specification of element types. DCDs (Bray et al., 2000) and XML-Schemas (Microsoft, 2001) are proposed to be extensions to DTDs to support these. If DCDs and XML Schemas become standard, then in order to have a generally usable application, the additional information on data types would aid in our generation process. We will further extend our work to deal with DCDs or XML Schema in the future.

Acknowledgements

The authors wish to thank the anonymous referees whose invaluable comments helped to improve this paper substantially. This work was partially supported by National Science Council, Taiwan, ROC under contract no. NSC-88-2416-H-327-010.

References


