Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica

OMZETTEN VAN CONCEPTUELE SCHEMA'S NAAR XML DOCUMENT TYPE DEFINITIONS

Kurt Lommens
Academiejaar
2000-2001

Verhandeling voorgedragen tot het behalen van
de graad van Licentiaat in de Toegepaste Informatica

Promotor : Prof. Dr. Olga De Troyer

Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica

MAPPING CONCEPTUAL SCHEMA'S TO XML DOCUMENT TYPE DEFINITIONS

Kurt Lommens
Academic Year
2000-2001

Essay brought forward to achieve the degree
of Licentiate in the Applied Computer Sciences

Promotor : Prof. Dr. Olga De Troyer

Samenvatting

In de wereld van website ontwikkelaars (autheurs en site designers) heeft HTML (HyperText Markup language) lang een quasi monopoly gehad als de standaard taal voor het publiceren op het wereldwijde web. De taal werd oorspronkelijk ontwikkeld als een eenvouding en makkelijk aan te leren middel om websites aan te maken via een statische set van voorgedefinieerde tags en attributen die toelaten om structuur en inhoud van sites te markeren. Door de steeds stijgende complexiteit van de onderliggende software systemen is tegenwoordig echter de nood ontstaan voor een krachtiger standaard.

Op dat vlak lijkt voor XML (the eXtensible Markup Language) een mooie toekomst weggelegd te zijn. In tegenstelling tot HTML maakt XML een onderscheid tussen gegevens, structuur en stijl. In deze thesis zullen we ons voornamelijk richten op structuur voor XML documenten. We zullen een inspanning beschrijven om ORM (Object-Role Modeling), een techniek geleend uit de wereld van database ontwikkelaars, te integreren in het design proces van XML DTD's (Document Type Definitions).

Alhoewel we tijdens deze integratie een paar belangrijke problemen zullen blootleggen, zullen we ook het belang van onze inspanning kunnen aantonen in het kader van de toekomstige ontwikkeling van websites. Zowel ontwikkelaars van websites als ontwikkelaars van databases hebben een aanzienlijk belang bij een dergelijke integratie: Het laat website ontwikkelaars niet alleen toe om te beschikken over een techniek om de structuur van hun XML documenten te modeleren op conceptueel niveau, het laat ook database ontwikkelaars toe om hun conceptuele schema's te gebruiken voor meer dan enkel het ontwerp van genormaliseerde database tabellen.

Nochtans zal deze thesis aantonen dat de huidige XML syntax eigenlijk te arm is om een echt succesvolle integratie met ORM te garanderen. Een mogelijke oplossing voor dit probleem is natuurlijk het uitbreiden van de XML syntax. De inspanningen van de *XML-Schema Working Group* binnen het kader van het World Wide Web Consortium (W3C) zijn hiervan een goed voorbeeld. Het kan echter ook interessant zijn voor onderzoekers om andere conceptuele modeleer-technieken te gebruiken (bv. OMT of UML).

In ieder geval, wat er ook gebeurt op dat vlak, het onderliggende idee zal altijd hetzelfde blijven: technieken die ontstaan zijn door ervaring in het ontwikkelen van software systemen moeten ook toegepast worden bij het ontwerpen van websites. In feite vormt het integreren van conceptuele modeleer-technieken en standaarden voor website ontwikkeling maar één beeld in het volledige verhaal. Het integreren van scripting talen zoals JavaScript en VBScript in web browsers is bijvoorbeeld een stap in dezelfde richting.

Abstract

In the world of web content developers (page authors and site designers), the HyperText Markup Language (HTML) has been widely regarded as as the standard publishing language of the *World Wide Web* for many years. It was originally conceived as a simple and easy-to-learn means to create web pages, providing a static set of pre-defined tags and attributes to markup structure and content. However, as the complexity of underlying software systems has increased over the last years, the need for a more powerful standard has become obvious.

In this regard, the eXtensible Markup Language (XML) seems to be the wave of the future. Unlike HTML, XML makes a clear distinction between *data*, *structure* and *style* in website design. In this thesis, we will focus on *structure* for XML documents. We will describe an effort to integrate Object-Role Modeling (ORM), a Conceptual Modeling technique borrowed from the world of database developers, in the design process of XML Document Type Definitions (DTD).

Although we will find our effort to suffer from a few important drawbacks, we will find the integration of ORM and XML to be interesting for future web development. In fact, this thesis will show the importance of this integration exercise for both involved parties: web content and database developers. Not only does it allow web content developers to model structure for XML documents at the conceptual level, it also allows database developers to use their conceptual schema's for more than just creating normalized database tables.

However, this thesis will reveal the current XML syntax to be too poor in order to ensure fully successful integration with ORM. The enrichment of the XML syntax by standardized extensions might be a fair solution to this problem. The efforts made by the *XML-Schema Working Group* at the World Wide Web Consortium (W3C) constitute a good example in this field. On the other hand, it might be interesting for researchers to integrate other Conceptual Modeling techniques, such as OMT or UML, in XML (or extensions to it) in the near future.

Anyhow, whatever the future brings, the basic objective will always remain the same: techniques developed through experience in the engineering of software systems must also be applied to web development. The integration of Conceptual Modeling techniques and web development standards constitutes only one image in the complete story. In fact, the incorporation of scripting languages, such as JavaScript or VBScript, in web browsers constitutes a step in the same direction.

Acknowledgement

I would like to thank the following people for their contributions to my studies and to this document:

- **Prof. Dr. Olga De Troyer**

  For the critical remarks and corrections on my draft versions. They have been very inspiring and have contributed a great deal to improve to quality of this document.

- **Prof. Dr. Robert Meersman**

  For the introduction to Object-Role Modeling (ORM) during the previous academic year and his efforts to obtain the VisioModeler file format.

- **Prof. Dr. Dirk Vermeir**

  For the introduction to the C++ programming language. I have hasseled him with a lot of questions and programming problems during the last two academic years. I would like to thank him for his efforts and patience in trying to find understandable solutions.

- **My family and friends**

  For trying to understand my complete absence and bad moods for the period of my education at the Free University of Brussels.

Table of Contents

MAPPING CONCEPTUAL SCHEMA'S TO XML DOCUMENT TYPE DEFINITIONS

Abbreviations

API	Application Programming Interface
ASP	Active Server Pages
CASE	Computer Aided Software Engineering
CORBA	Common Object Request Broker Architecture
CS	Conceptual Schema
CSDP	Conceptual Schema Design Procedure
CSS	Cascading Style Sheets
DCD	Document Content Description
DCOM	Distributed Component Object Model
DDML	Document Definition Markup Language
DFD	Data Flow Diagram
DHTML	Dynamic Hypertext Markup Language
DOM	Document Object Model
DTD	Document Type Definition
EBNF	Extended Backus Naur Form
ERD	Entity-Relationship Diagram
ERM	Entity-Relationship Modeling
FORML	Formal Object-Role Modeling Language
HTML	Hypertext Markup Language
IS	Information System
LOT	Lexical Object Type
MOT	Mayor Object Type
NIAM	Natural-language Information Analysis Method
NOLOT	Non-Lexical Object Type
OMG	Object Management Group
OMT	Object Modeling Technique
OOA&D	Object-Oriented Analysis and Design Methods
ORM	Object-Role Modeling
PI	Processing Instruction
RIDL	Reference and Idea Language
SGML	Standard Generalized Markup Language
SOX	Schema for Object-Oriented XML
SQL	Structured Query Language
STL	Standard Template Library
UML	Unified Modeling Language
UoD	Universe of Discourse
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WWW	World Wide Web
XLink	XML Linking Language
XSL	eXtensible Stylesheet Language
XML	eXtensible Markup Language
XPath	XML Path Language
XPointer	XML Pointer Language
XQuery	XML Query Language

Introduction

Nowadays software systems have a strong tendency towards more complexity. The same goes for web-related systems. In the context of the internet, the technical limitations of the HyperText Markup Language (HTML) seem to feed impressive efforts made by academic researchers and leading software companies to construct more powerful standards.

One of these new standards, the eXtensible Markup Language (XML), promises to be the wave of the future for web development and structured data transfer. Unlike HTML, the XML standard makes a clear distinction between *data*, *structure* and *style*. XML allows the storage of data according to a well-defined, fully adaptable structural format and freed from style issues. Storing data this way clearly makes them very suitable for easy transfer and further processing.

In this thesis, we will introduce the XML Document Type Definition (DTD) as the basic means to define data structure for XML documents. In order to simplify and automate the construction of sometimes complex DTD's, we will lean back on a conceptual modeling technique borrowed from the world of relational databases: Object-Role Modeling (ORM).

ORM has become increasingly popular as a conceptual modeling technique for the construction of normalized, relational database tables. ORM typically views the world in terms of objects playing roles. It includes various design procedures to help modelers develop Conceptual Schema's (CS). These CS's can be easily validated and mapped to database schema's by the use of an ORM tool, such as VisioModeler.

Using CS's to model data structure for XML documents does not only simplify the construction of XML DTD's, it also allows database developers to use the same CS to construct both normalized database tables and data structures for websites. In this regard, the value of existing CS's will be significantly increased.

In order to create this important link between ORM and XML, we will elaborate an algorithm that maps a maximum of syntactic and semantic properties from a CS to a DTD. We will also develop a program, TransORM, to test and demonstrate the feasability of our algorithm.

The structure of this thesis is as follows: A short introduction to Conceptual Modeling and ORM is presented in chapter 1. In chapter 2, we will focus on XML and XML-related work. Chapter 3 presents our mapping algorithm and chapter 4 describes the TransORM program. Finally, chapter 5 presents general conclusions.

1. Conceptual Modeling

1.1 Introduction

Throughout history, the human race has been using a wide variety of techniques to model things from *the real world*. All of these techniques are based on abstraction. Real world objects are mapped to abstract representations within the context of one (or more) model(s). Abstract representations make it easier to describe attributes and (complex) relationships between the real world objects they represent, even if those objects are non-material. Of course, different techniques can represent the same real world object in a completely different manner. In order to understand the meaning of abstract representations, one must be familiar with the context they are used in.

A good example of people using abstraction to represent real world objects is human language. Within the context of one human language, most real world objects get a unique representation in the form of a word or a group of words, which is no more than a unique combination of abstract linguistic symbols. Through education, people are taught the correspondence between (abstract) words and the real word objects they refer to in that particular language.

The link between real world objects and abstract representations within a particular model is sometimes a complex matter. This has been demonstrated in some of the paintings and essays of René Magritte, a Belgian surrealist painter and philosopher in the early 20th century. One example is the painting in Figure 1.1. Magritte tries to explain with it that people need to be familiar with the material notion of a pipe and both its graphical and linguistic representation in order to understand all the abstract links on the painting. But even then, they can never be sure if the represented object materially exists. The image could be the artist's rendering of a real pipe. It could also be completely made up by the artist. One can not tell the difference just by looking at the image. Michel Foucault[1], a philosopher who wrote a book on this painting, resumed his conclusions in some phrases like “this is not a pipe but the image of a pipe”, “this painting is not a pipe”, “the text “ceci n’est pas une pipe” is not a pipe”, etc.

Figure 1.1: René Magritte, “Ceci n’est pas une pipe”

[1] A good book on this subject is: James Harkness, 1983, *This is not a pipe*, Berkeley, University of California Press. This version holds illustrations and letters by René Magritte and has significant variations from the original version in *Dit et Ecrits* from Michel Foucault.

In the computing world, we have to deal with the same kind of treachery of representations. Images of files, processes and network connections do not show the real bits in memory, packets or disks. They are nothing but complex, abstract representations produced by layer upon layer of hardware and software. Everyone with access to one or more of these layers can possibly damage the correct production of the final image. Therefore, computer representations should always be handled with care.

Now that we have pointed out the possible danger of using abstract representations, we can go back to our main thing: What is conceptual modeling all about ?

1.2 The value of Conceptual Modeling

Conceptual modeling is a method to create a formal model in which every entity being modeled in the real world has a transparent and one-to-one (unique) correspondence to an object in the model. A conceptual modeling language, like an object-oriented language, encapsulates all of the information about a real world entity (including its behavior) in the object itself. A conceptual modeling language goes beyond a standard object-oriented language by replacing the simple instance variables with attributes that encapsulate integrity constraints and the semantics of relationships to other objects. Because conceptual modeling languages map directly from entities in the real world to objects in the computer-based model, they make it easier to design and implement systems. The resulting systems are easier to use since they are semantically transparent to users who already know the problem domain[2].

Apart from the technical details of conceptual modeling, we could ask ourselves why we need this kind of modeling in the creation process of software systems. The short answer is quality. As the complexity of software systems increases exponentially, the need for overview and (good) design grows along. Decades of software quality-assurance research have revealed certain axioms. A major finding was that most application flaws and bugs are the result of a deficient design. This is the part where conceptual modeling reaches a helping hand: it enables software engineers to implement their systems based upon a transparent design method.

In this thesis, we will introduce and use Object-Role Modeling (ORM) as a conceptual data modeling technique to model structure of data in websites (cfr. section 2 and 3). Why do we prefer ORM over older methodologies like Data Flow Diagrams (DFD) and Entity-Relationship Modeling (ERM) ?

1.3 Object-Role Modeling versus other data modeling methods

The first generations of modeling tools were centered on process and data modeling. One of the early data modeling techniques was the Natural-language Information Analysis Method (NIAM). This method was largely developed by Dr. Eckhard Falkenberg and Dr. Shir Nijssen, with significant contributions from Professors Dr. Robert Meersman (added support for subtyping), Dr. Dirk Vermeir and Dr. Olga de Troyer.

---

[2] Gary F. SIMONS, 1994, *Conceptual modeling versus visual modeling: a technological key to building consensus*, Dallas, Summer Institute of Linguistics – A paper presented at: Consensus ex Machina, Joint International Conference of the Association for Literary and Linguistic Computing and the Association for Computing and the Humanities, Paris, 19-23 April 1994 - http://www.sil.org/cellar/ach94/ach94.html

The basic NIAM philosophy is that information analysis intends to model the communication about a certain Universe of Discourse (UoD) but does not intend to model the UoD itself. Consequently all existing elements in NIAM must be consistent with this philosophy[3]. NIAM has been subject to improvements and extensions resulting in modified versions such as Communication Oriented NIAM (CO-NIAM) and Fully Communication Oriented NIAM (FCO-NIAM). ORM also fits in this image as a modeling approach based on extensions to NIAM. It constitutes a conceptual modeling approach that views the world in terms of objects and the roles they play. ORM was mainly initiated by Dr. Terry Halpin[4] as a fundamental design method for physical relational database schema's. The term *Object-Role Modeling* was originally used by Dr. Eckhard Falkenberg for his modeling framework and is now used generically to cover the various versions of the modeling approach.

A good alternative to ORM and popular approach to conceptual modeling is Entity-Relationship Modeling (ERM). The method was introduced back in 1976 by Dr. Peter Chen[5]. It describes a process of creating Entity-Relationship Diagrams (ERD). A completed ERD can be used either individually either in combination with Data Flow Diagrams to provide a comprehensive Information System (IS) logical design. Since its introduction, ERM has been considered as a valuable design method and became increasingly popular among database developers.

The process of creating an ERD can be described as follows:

Steps
1. Identify the entities
2. Determine the attributes for each entity
3. Select the primary key for each entity
4. Establish the relationships between the entities
5. Draw an entity model
6. Test the relationships and the keys

Table 1.1: the process of creating Entity-Relationship Diagrams

In recent years, ORM seems to gain importance over ERM (and other data modeling techniques). As stated by Dr. Halpin, ORM diagrams are typically more expressive and can be populated with instances allowing validation with the client using natural language. Obviously, validation modalities help to improve the overall quality and usability of the design procedure. Next to this, ERD's can be easily and safely abstracted from ORM diagrams. In addition, the introduction of VisioModeler[6] as a CASE (Computer Aided Software Engineering) tool supporting conceptual modeling through ORM has been a major advantage in the drive of ORM towards success.

[3] G. P. Bakema, J. P. C. Zwart, H. van der Lek, 1993, *Fully Communication Oriented NIAM,* paper presented at http://www.fcoim.com/FCONIAM.HTM

[4] Terry HALPIN, 1999, *Conceptual Schema & Relational Database Design* (Second Edition), WytLytPub, Bellevue, USA

[5] Peter Chen, March 1976, *The Entity-Relationship Model -- Toward a Unified View of Data,* ACM Transactions on Database Systems, Vol. 1, No. 1, pages 9 - 36

[6] The Visio Corporation has been taken over by Microsoft in 2000. This has resulted in a new software package called Visio2000 which is sold by Microsoft as an extension to their Office2000 package.

1.4 Basic Object-Role Modeling

1.4.1 The Conceptual Schema Design Procedure (CSDP)

In his book, Dr. Halpin describes a formal procedure for designing a Conceptual Schema (CS) from a given UoD. This procedure is commonly known as the Conceptual Schema Design Procedure (CSDP):

Steps
1. transform familiar information examples into elementary facts and apply quality checks
2. draw the fact types and apply a population check
3. check for entity types that should be combined and note any arithmetic derivations
4. add uniqueness constraints and check the arity of fact types
5. add mandatory role constraints and check for logical derivations
6. add value, set comparisons and subtyping constraints
7. add other constraints and perform final checks

Table 1.2: the Conceptual Schema Design Procedure (CSDP)

STEP 1: *transform familiar information examples into elementary facts and apply quality checks*

In order to model a Universe of Discourse (UoD), we need people familiar with it. UoD experts, also called *subject matter experts*, are needed to (help) verbalize the UoD to be modeled in order to avoid misinterpretation. Consider the example of an academic domain. Only people familiar with the particular academic domain can explain in detail how it is organized. The knowledge of UoD experts, sometimes combined with information derived from existing or required input/output forms, constitutes a solid working base to produce a reliable verbalized version of the UoD.

Throughout this thesis, we will use the example of a simplified academic domain for reasons of demonstration. The verbalized description of this sample UoD could be the following:

Every member of the academic domain is either a professor or a student and has at least:

- *An email address. This address is unique for every member.*
- *A name. Different members can have the same name.*
- *A date of birth. Each date is referenced by a unique combination of a daynr, a monthnr and a yearnr. Different members can have the same date of birth.*

A professor always holds a unique professorID. A professor can teach a course.

A student always holds a unique studentID. A student must follow at least one course.

Courses hold a unique code and must be teached by exactly one professor. One single course can be followed by different students.

The information in this verbalized UoD is stated in terms of *elementary facts*. Basically, identifying elementary facts means describing objects holding properties or participating in a relationship. More formally, an elementary fact can be defined as a relationship (role) between two or more object types, where that relationship cannot be expressed as a conjunction of simpler facts. In our sample case, saying that a member has an email address constitutes an elementary fact.

ORM makes a distinction between two types of objects[7]: Lexical Object Types (LOT) and Non-Lexical Object Types (NOLOT). LOT's are character strings or numbers: they are identified by *constants* (e.g. David R. Williams, 25899). NOLOT's are real world objects that are identified by a definite description (e.g. the student with studentID 25899). Definite descriptions typically indicate the NOLOT (e.g. student), a value (e.g. 25899) and a reference mode (e.g. studentID). A reference mode is the manner in which a LOT refers to a NOLOT. During Step 1 of the CSDP, constant quality checks should be applied to ensure that all objects are well identified.

STEP 2: *draw the fact types and apply a population check*

For reasons of brevity, we will not draw the complete draft diagram at this point. We can illustrate this step of the CSDP by picking just a few of the fact types. Consider the following partial draft diagram:

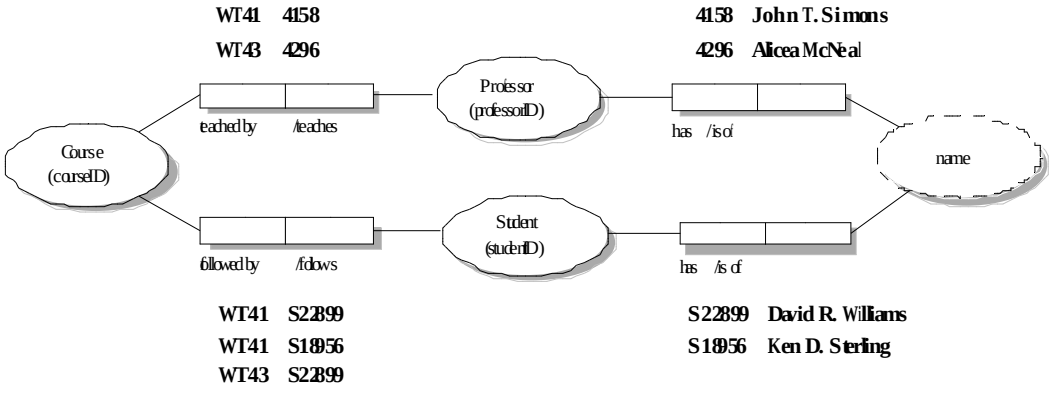


Figure 1.2.: a partial draft diagram for a sample academic domain

This diagram holds four fact types. All fact tables are populated with sample data from our UoD. This test population serves only as a means of validation and is therefore not an integral part of the diagram. This way of validating all fact types and possible combinations of fact types ensures a better quality of the final CS. In this phase, possible errors can be easily identified by finding real world data that do not conform the drawn diagram. Once we have a solid draft diagram, we can move to the next step in the CSDP.

[7] Dr. Halpin calls them respectively *Value Types* and *Entity Types*.

STEP 3: *check for entity types that should be combined and note any arithmetic derivations*

In our sample case, an academic member is either a professor or a student. The set of all professors and students equals the set of all members. We can express this in our (draft) diagram by introducing NOLOT *member* and specifying both NOLOT *professor* and NOLOT *students* to be subtypes of it (cfr. also Step 6). A member can be identified by the LOT *email*, since email addresses are unique for all people in the academic domain.

The introduction of NOLOT *member* causes valuable information to be lost from our diagram: Is a member a professor or a student? To solve this, we introduce a new fact type: *member has membercode*. We will use *P* for professor and *S* for student. The diagram drawn in Figure 1.2 will be modified like this:

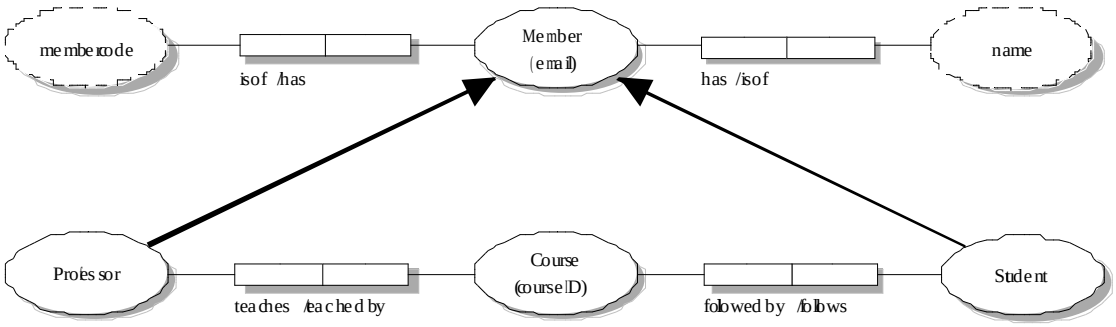


Figure 1.3: adapted partial draft diagram for a sample academic domain

Experienced ORM users will argue that there is no need to maintain NOLOT *professor* and NOLOT *student* in our diagram. This is of course true. By introducing the LOT *membercode*, we can at any time find out whether a member is a professor or a student. We have only introduced the subtypes in our diagram for later use (cfr. section 3).

Sometimes, it is possible to derive fact types from others by arithmetic. Suppose we would want to record the number of students following a particular course. This number can be obtained by simple counting. Such fact types are called *derivable*. We can insert derived fact types in our diagram by using the asterisk (*) followed by a derivation rule. For example (*iff* stands for *if and only if*):

> * Course *c* is followed by students in Quantity *q* **iff** *q* = **count each** student **who** follows Course *c*

STEP 4: *add uniqueness constraints and check the arity of fact types*

Uniqueness constraints are used to assert that entries in one or more roles occur there at most once. The term *uniqueness constraints* is just another naming for *identifier constraints* in NIAM. Throughout this document, we will use the NIAM naming conventions. Identifier constraints are either internal or external.

Internal identifier constraints are marked by placing arrows over the roles in fact types. An arrow spanning n roles of a fact type (n>0) indicates that each corresponding n-tuple in the associated fact table is unique. This is shown in Figure 1.4:

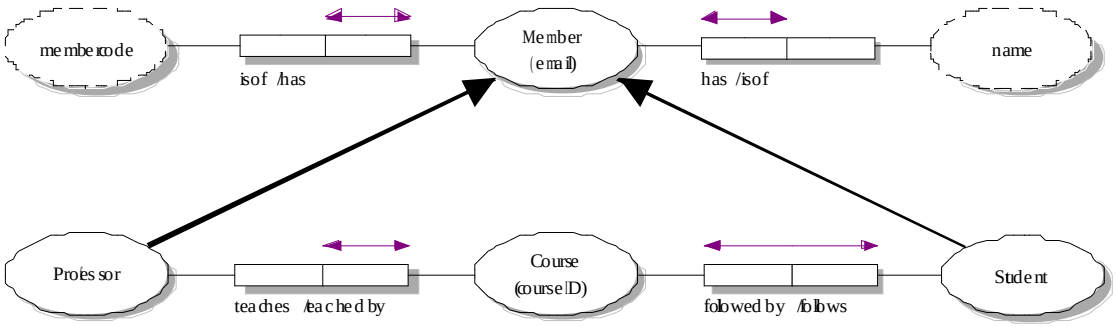


Figure 1.4: partial draft diagram for a sample academic domain with identifier constraints

The arrow over the role *teached by* in the fact type between NOLOT *professor* and NOLOT *course* means that no duplicates are allowed for instances playing that role. In other words, a course can only be teached by one professor. Since there is no arrow over the role *teaches*, a professor can teach many courses.

Note the difference with the arrow over the roles in the fact type between NOLOT *student* and NOLOT *course*. This arrows spans both roles, indicating that a student can follow more than one course and that a course can be followed by more than one student. However, each combination in the fact type is unique. Obviously, there's no point in a student following the same course twice and/or a course followed by the same student twice.

External identifier constraints span roles of different fact types. They are indicated using a circled *u* on the diagram: this specifies the combination of connected roles to be unique in the natural join of the fact types. If we take the example of NOLOT *date*, we find that each date is identified by a unique combination of numbers indicating a day, a month and a year. We can show this by using an external identifier constraint:

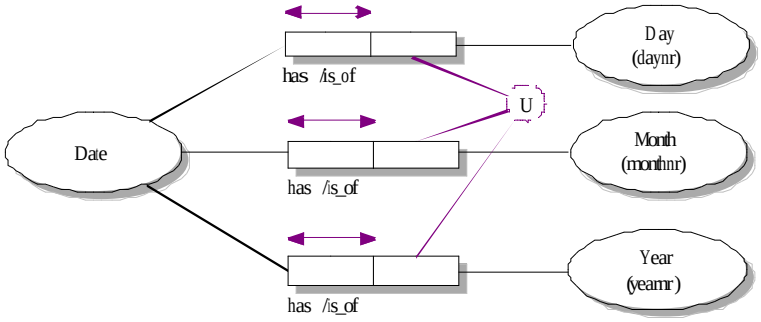


Figure 1.5: draft diagram for date, holding an external identifier constraint

Experienced ORM users will argue that NOLOT's *day, month* and *year* are just numbers and should therefore be indicated as LOT's. They are right once again. We have made them NOLOT's for reasons of demonstration in section 3.

Once identifier constraints have been added to the diagram, an arity check can be performed. This check is included to eliminate splitable fact types from our diagram. Since we only intend to provide the reader of this document with the basics of ORM, we leave out arity checks.

STEP 5: *add mandatory role constraints and check for logical derivations*

A role is mandatory (total) if each and every instance of an object must play that role. This is indicated by a placing a dot where the role connects with its object type. Looking at Figure 1.6, we can say that each member must have a name and a membercode, each student must follow a course and each course must be teached by a professor.

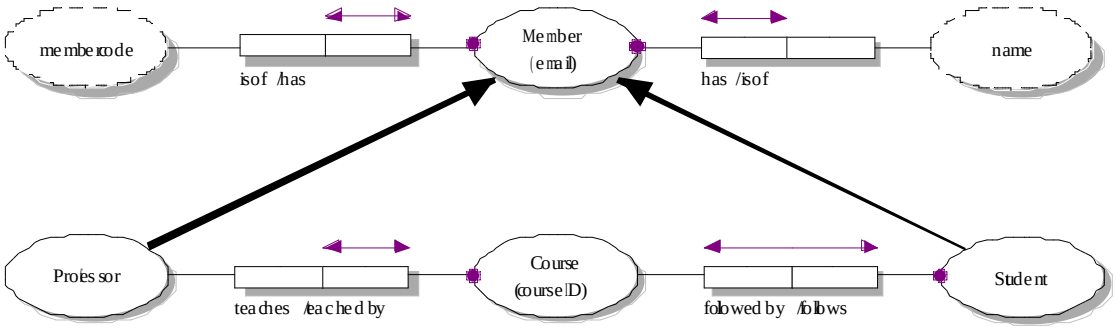


Figure 1.6: partial draft diagram for a sample academic domain with mandatory role constraints

For reasons of brevity, *logical derivations* will not be used in our sample diagram.

STEP 6: *add value, set comparisons and subtyping constraints*

Value constraints specify a list of possible values for a LOT. These usually take the form of an enumeration or range and are displayed in braces beside the LOT. For example, LOT membercode can only have two possible values: *P* and *S*. This is indicated by putting *{ 'P','S' }* beside *membercode* in the diagram.

Set comparison constraints specify subset, equality or exclusion constraints between compatible roles or between sequences of compatible roles. Roles are compatible if they are played by the same object type (or by object types with a common supertype). Set comparison constraints will not be used in our sample diagram.

On the other hand, subtyping constraints are an important issue in our case. As stated in the verbalized UoD, an academic member is either a professor or a student. This means that the set of all professors and all students equals the set of all members. In ORM, we can indicate this by placing a totality constraint between the subtypes of supertype *member*. This is visualized using a circled dot. In addition, a student can not be a professor and a professor can not be a student. In other words, the set of professors and the set of students are mutually exclusive. This is indicated using a circled cross between the two subtypes. Both constraints will not be shown on the final diagram since VisioModeler, the drawing tool used to create the diagram, did not support visualization of subtyping constraints at the time of writing this text.

STEP 7: *add other constraints and perform final checks*

To improve the quality of our diagram even further, formal ORM provides a wide variety of other possible constraints to be added, such as:

- Role frequency constraints
- Ring constraints for binary facts
- Cardinality constraints
- Path implication and path equivalence
- State transition constraints
- General RIDL[8] procedural constraints
- ...

These constraints will not be discussed in this document.

The CSDP ends with some final checks to ensure that the diagram is consistent with the original examples, avoids redundancy and is complete.

1.4.2 The final Conceptual Schema

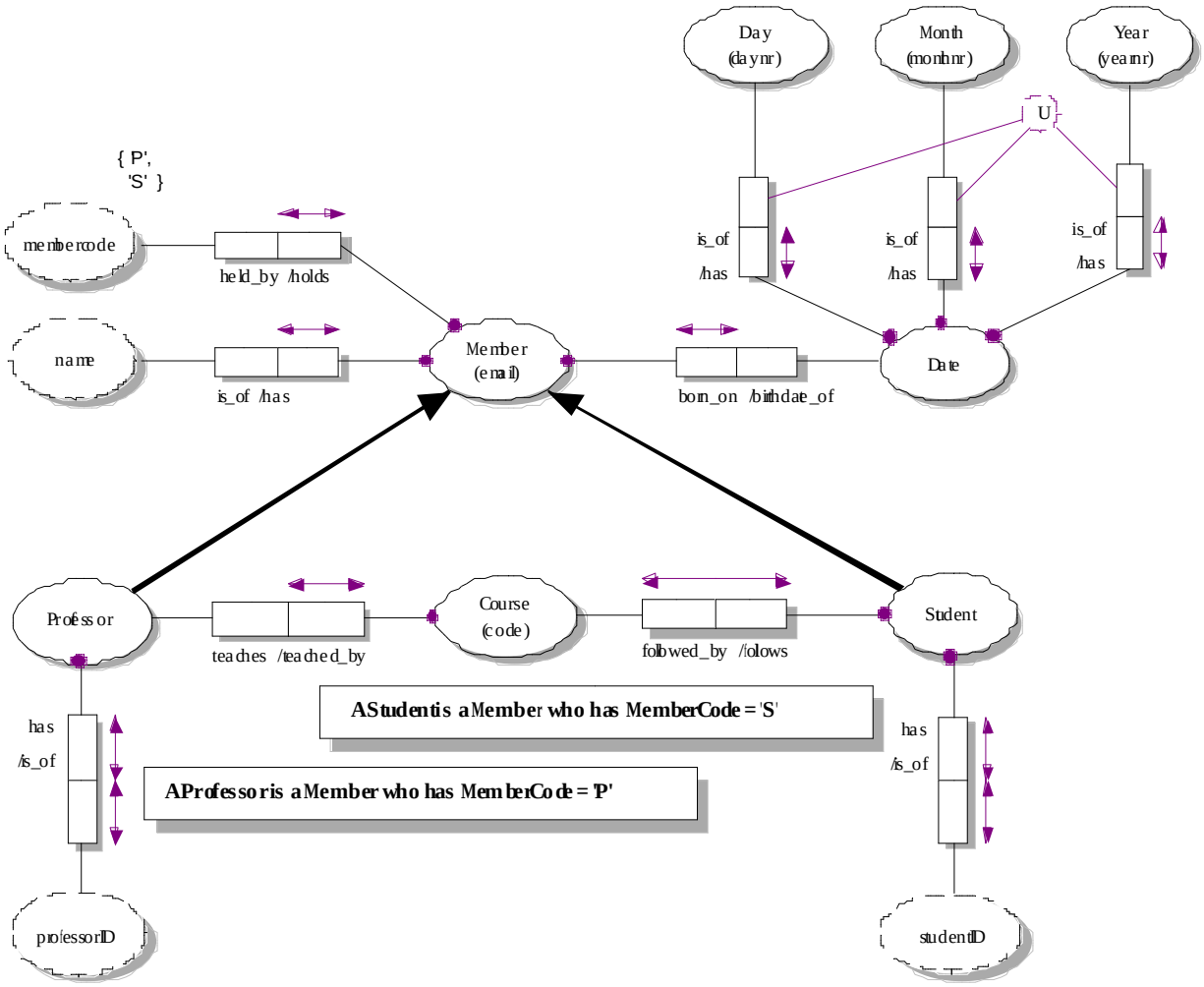


Figure 1.7: completed diagram for a sample academic domain

[8] Reference and Idea Language

The above diagram can be seen as a graphical representation (model) of the UoD. It is used as a somewhat hidden interface to the Formal Object-Role Modeling Language (FORML). Instead of creating text files containing FORML statements, developers are encouraged to create these graphical representations of the UoD. Obviously, graphical models tend to offer a much better overview and understanding of structure and relationships than encoded text files. Although, not everyone is convinced that graphical tools are the best solution for modeling[9].

1.4.3 Creating a physical database schema

As stated earlier, describing the UoD and creating a CS based on that description are only the two first steps in relational database design. In order to create physical (relational) database schema's, we need a formal mapping algorithm that uses CS's as input. Due to the inherent richness of ORM, most of the constraints on ORM diagrams will be lost in the process of creating physical database schema's. Designers using ORM in the design process of relational databases will need to incorporate these lost constraints in application software, which serves as a filter for all state changing database operations.

For our sample CS, using the mapping algorithm presented in Appendix 1 will result in the following normalized tables:

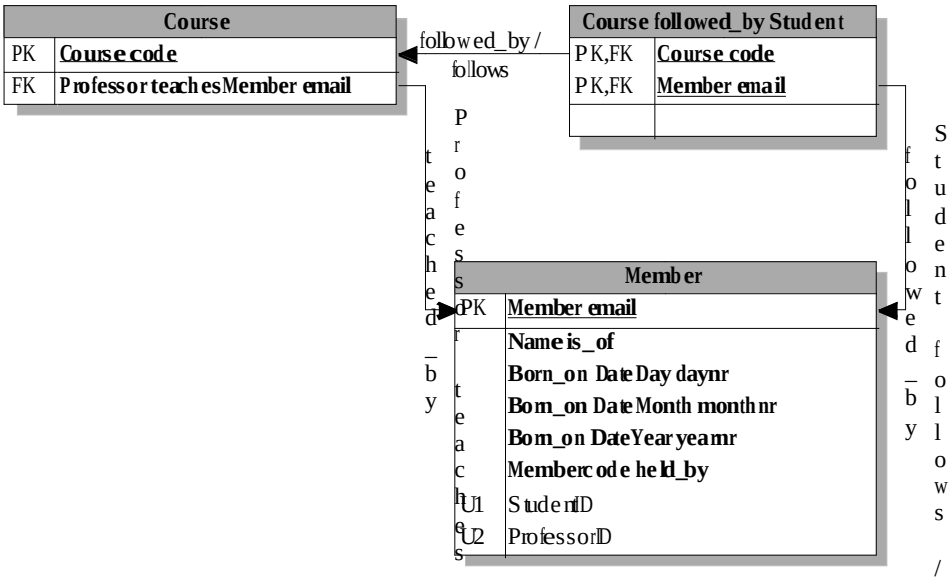


Figure 1.8: resulting normalized database tables.

[9] J. C. Nordbotten, Martha E. Crosby, *Recognising Graphic Detail - An Experiment in User Interpretation of Data Models*, 13th British National Conference on Databases, BNCOD 13, Manchester, United Kingdom, July 12-14, 1995, Proceedings, Lecture Notes in Computer Science, Vol. 940, Springer, ISBN 3-540-60100-7

1.5 Beyond Conceptual Data Modeling

So far, we have limited our approach towards conceptual modeling to data modeling, using ORM as the design method. Next to data modeling, at least two different modeling techniques are believed to gain importance in the years to come: object modeling and content modeling.

In the world of Object-Oriented programming languages, the Unified Modeling Language (UML) has become very popular as a language for object modeling. Although UML goes beyond the scope of this thesis, it deserves a few explanatory words.

The UML has been created as the successor of a series of Object-Oriented Analysis and Design methods (OOA&D) dating from the late eighties and the early nineties. It combines separate work from Grady Booch, Jim Rumbaugh[10] and Ivar Jacobson. The UML supports several different views of a system - class diagrams, behavior diagrams, use-case diagrams and implementation diagrams. It is constructed as a standard modeling language, not as a method. The language has been subject to a standardization procedure by the Object Management Group (OMG) and has now become an OMG-standard[11].

Although the UML is a powerful and interesting (object) modeling language, we will not use it in the scope of this thesis. Instead, we will focus on the integration of data modeling and content modeling techniques. Throughout this chapter, we have introduced and explained ORM as a powerful data modeling technique. In the next chapter, we will introduce the eXtensible Markup Language (XML) as an interesting programming language for modeling the content of websites.

[10] mainly known as the major force behind the Object Modeling Technique (OMT)

[11] Martin Fowler, Kendall Scott, *UML destilled: applying the standard object modeling language*, Addison-Wesley, 1997

2. The eXtensible Markup Language (XML)

2.1 Introduction

The Internet has revolutionized the computer and communications world like nothing before. Within a few decades, it has become the single most important means of communication around the world. Throughout its exponential growth process, it has given birth to a wide variety of new computer-related economies, bringing the computer in hundreds of millions households and companies. Although its growth seems to stagnate a little bit in recent years, the internet is believed to continue its path of worldwide success in the years to come.

Due to this massive use of the internet, method standardization has become an important issue. In the context of the World Wide Web (WWW), which is a fundamental part of the Internet, the HyperText Markup Language (HTML) has been the only standard that was publicly available for many years. HTML is an application format based on the Standard Generalized Markup Language (SGML), both defined by the World Wide Web Consortium (W3C). In fact, HTML has been created as a simple and easy-to-learn means to create web pages, providing a static set of pre-defined tags and attributes to markup structure and content.

Over the last few years, a multitude of new standards have become available in this field: Dynamic HTML (DHTML), Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), Active Server Pages (ASP) and many more. However, another standard, which has been around for half a decade, seems to be the wave of the future: the eXtensible Markup Language (XML). XML has been created as a clear subset of SGML. It was mainly developed to be less complex than SGML and to be more powerful than HTML. The W3C predicts that HTML, XML and SGML will all be used in the future, none makes the others obsolete.

2.2 The need for XML

As mentioned in section 1, nowadays software systems have a strong tendency towards more complexity. The same goes for web-related systems. Considering the static nature of HTML, its technical capabilities do not suffice in most cases for more advanced web pages and applications. Many developers have switched to SGML for more advanced publications, but this standard has proven itself to be too large and too complex to gain public acceptance.

The W3C answer to this problem was XML. This standard was developed to combine the strengths of both SGML and HTML, eliminating their weaknesses as much as possible. In order to fulfil this rather ambitious intention, the W3C has set up some clear design goals for the development process of XML. This next list is taken directly from the XML 1.0 Specification (second edition), which was adopted as an official Recommendation by the W3C on the 6th of November 2000:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

Table 2.1: the W3C design goals for XML

One of the main features of XML is that the standard makes a clear distinction between data, structure and style. In fact, XML documents are data-based and do therefore not contain any information about structure[12] or style. This makes it possible for different people to define their own views of how data should be represented. Structure and style information for XML documents are typically stored in separate documents.

2.3 XML documents are data-based

XML documents contain structured data built up using markup tags. Unlike HTML tags, XML tags have no pre-defined meaning. The definition and structure of XML tags is left to Document Type Definitions (cfr. section 2.4.), while style (or displaying) issues are left to stylesheet languages (cfr. section 2.5). Both are fully adaptable to the personal needs of people creating the XML documents.

The following example might make clear the difference between HTML and XML tags:

```
<H1>Academic Members</H1>
<BR>
<TABLE WIDTH=”100%” BORDER=”0”>
<TR>
<TD>Email</TD>
<TD>david.williams@vub.ac.be</TD></TR>
<TR>
<TD>Name</TD>
<TD>David R. Williams</TD></TR>
<TR>
<TD>Membercode</TD>
<TD>S</TD></TR>
<TR>
<TD>Birthdate</TD>
<TD>07/05/1978</TD></TR>
<TR>
<TD>StudentID</TD>
<TD>S25899</TD></TR>
<TR>
<TD>Follows</TD>
<TD>Logic Programming</TD></TR>
</TABLE>
```

Table 2.2: a HTML code fragment describing a member entry for an academic domain

[12] this is technically not true when internal DTD’s are used.

All tags in this HTML code fragment are style-related. In others words, they determine how the data will be displayed. Marking up data this way clearly makes it pretty complicated to use the raw data for any other purpose than just displaying them. In addition, the used tags are pre-defined in the formal HTML syntax. No other tags than those defined in the HTML syntax can ever be used in a HTML document. Next to this, the HTML tags do not provide any information about the data being marked up.

In an XML document, the same raw data could possibly be represented like this:

```
<AcademicMembers>
        <Student StudentID=”S25899”>
                <Email>david.williams@vub.ac.be</Email>
                <Name>David R. Williams</Name>
                <Birthdate>
                        <Day>07</Day>
                        <Month>05</Month>
                        <Year>1978</Year>
                </Birthdate>
                <Follows>Logic Programming</Follows>
        </Student>
</AcademicMembers>
```

Table 2.3: an XML code fragment describing a member entry for an academic domain

Clearly, none of the tags used in the above XML code fragment are style-based. They do not define or instruct in any way how the data will be displayed. The XML code fragment simply shows that raw data are marked up according to some structure format. However, this structure format is not defined in the code fragment itself. This makes XML marked up data very suitable for further use by any other application. Contrary to HTML tags, XML tags provide valuable information about the raw data being marked up, which of course increases transparency and readability of XML documents. It should also be noted that unlike HTML tags, XML tags are case sensitive, which means that <Student> is not the same as <student>.

Well-formed XML documents are defined as being in the form of a simple hierarchical tree. Each document has a root node, also known as the *document entity* or the *document root*. This root node may contain processing instructions (PI) and/or comments. It is the point of attachment for the document’s description using a DTD and it will always contain a sub-tree of XML elements. The root of this sub-tree is called the *document element*. All other elements in an XML document are descendants (children) of this document element. A formal XML document structure is presented in Figure 2.4, in which only the Prolog and the Epilog are optional:

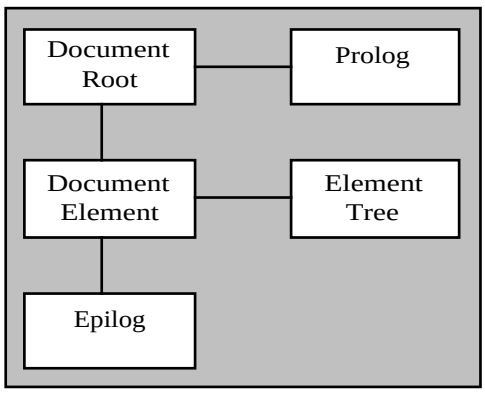


Figure 2.4: the formal XML document structure

As for the element tree, XML imposes proper nesting as a key constraint. This implies that no overlapping tags are allowed in well-formed XML documents. Unlike HTML parsers, XML parsers will produce a *not well-formed* error whenever an improperly nested tag is encountered.

Let us now look at the basic data structures that appear in XML documents.

2.4 XML and structure: the Document Type Definition (DTD)

DTD's are an integral part of the XML 1.0 Recommendation, allowing designers to communicate the syntactic rules of their vocabularies so that any other user of XML can interprete the structure of XML documents that conform to these vocabularies. In short words, DTD's define the structure of all data instances in XML documents. Whenever a data instance does not conform the syntactic rules defined in the DTD's linked to the document, it causes the document to be invalid. DTD's can either be internal or external to an XML document. Note that internal DTD's, also known as *internal subsets*, have priority over external DTD's, also known as *external subsets*, whenever name collision is encountered by the XML processor.

Internal DTD's are very useful. Since they are defined in the XML document itself, one never has to worry about finding the right DTD. However, they tend to add substantial size to the document whenever the defined vocabularies are getting complex. In addition, internal DTD's are more difficult to port to other documents than external ones. Common sense tells us to use internal DTD's only in those cases when the vocabularies are very simple or whenever we want to overrule or finetune declarations from external DTD's for particular document needs. In all other cases, external DTD's should be preferred over internal ones. A good design choice might sometimes be to use a mixture of internal and external DTD's.

The main link between XML documents and DTD's is the *DOCTYPE* tag, as shown in the following examples:

Link to an internal DTD:
<!DOCTYPE AcademicMembers [… internal subset decalarations …]>

Link to an external DTD:
<!DOCTYPE AcademicMembers SYSTEM "http://www.members.com/academic.dtd">

XML poses a key constraint on the naming after the the *DOCTYPE* tag. More specifically, the very first element (root) of a code fragment directly following one of the above *DOCTYPE* tags in an XML document must be *AcademicMembers*. In any other case, the XML processor will generate an error. For links to external DTD's, the location of the linked DTD is identified using an external identifier. Usually, this identifier consists of the keyword *SYSTEM*, followed by a Uniform Resource Identifier (URI). This URI can either point to a local location (hard-disk, CD-ROM, floppy-disk, etc), a network location or even an internet address.

2.4.1 XML Elements

The heart and soul of XML is the element. XML elements are defined in DTD's using the *ELEMENT* keyword. Elements can be considered as the XML containers for content. Formally, element content falls into four categories:

- **EMPTY** – an empty element has neither text nor child elements contained within it.

  ```
  <!ELEMENT Email EMPTY>
  ```

  Defining an element to be empty allows instances like this one in XML documents:

  ```
  <Email></Email>

  Or shorter:
  <Email/>
  ```

- **#PCDATA** – used to store parsed character data (or text).

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

  Defining the content of an element to be of type *PCDATA*, allows instances like this one in XML documents:

  ```
  <Email>david.williams@vub.ac.be</Email>
  ```

- **ANY** – used to store any type of content, provided that this content does not violate XML well-formed syntax. This content type is to be handled with care, as the XML processor will provide very little validity checking.

- ***Other elements*** – elements containing other elements (child elements) are called nested elements. In short words, nested elements hold other elements in their content model.

The *content model* of an element consists of a set of elements and/or the keyword #PCDATA, delimited by parentheses. Content models typically define element structure by the use of *order operators* and *cardinality operators*. Table 2.4 gives an overview of these operators. Examples of how they are used can be found in section 3.6.3.

Operator	Type	Usage
,	Order	Strict sequence
|	Order	Choice (logical OR)
?	Cardinality	Optional, may or may not appear
*	Cardinality	Zero or more
+	Cardinality	One or more
	Cardinality	Exactly one

Table 2.4: Order and Cardinality operators for XML content models

2.4.2 XML Attributes

Element tags can include one or more optional or mandatory attributes[13] that give further information about the elements they delimit. Attributes can only be specified in the start tag of an element or in an empty element tag, with the restriction that only one instance of an attribute name is allowed within the same tag. In the next code fragment, *StudentID* is used as an attribute for the nested element *Student*:

```
<Student StudentID="S25899">
        <Email>david.williams@vub.ac.be</Email>
        <Name>David R. Williams</Name>
        <Birthdate>
                <Day>07</Day>
                <Month>05</Month>
                <Year>1978</Year>
                </Birthdate>
        <Follows>Logic Programming</Follows>
</Student>
```

Attributes are declared in the DTD using the *ATTLIST* keyword. XML makes a clear distinction between three types of attributes:

- **String attribute types (CDATA)** – the value of the attribute consists of character data (string).

- **Tokenized attribute types** – the value of the attribute consists of one or more tokens that are significant to XML. There are seven types of tokenized attributes:

  - **ID** – this attribute serves as an identifier for the element. For those familiar with relational databases, an ID attribute functions much like a Primary Key. No two elements can have the same ID attribute value in a valid XML document. An important restriction is that each element can only have one single attribute of type ID (cfr. lower).
  - **IDREF** – this attribute is a pointer to an ID attribute, declared elsewhere in the XML document. In the world of relational databases, this concept matches the notion of a Foreign Key.
  - **IDREFS** – the value of this attribute consists of one or more attributes of type IDREF, separated by spaces.
  - **ENTITY** – this attribute is a pointer to an external entity, either declared in the internal or external subset (cfr. section 2.4.3).
  - **ENTITIES** – this value of this attribute consists of one or more Entity type values, separated by spaces.
  - **NMTOKEN** – the value of this attribute is a name token string consisting of any mixture of name characters.
  - **NMTOKENS** – the value of this attribute consists of one or more NMTOKEN type values, separated by spaces.

- **Enumerated attribute types** – the value of this attribute is simply a list of possible values. Each listed value has to be a valid *NMTOKEN*.

[13] Two special atttributes have been defined by the XML 1.0 Recommendation: *xml:space* and *xml:lang*. For further information about those two attributes, we refer to sections 2.10 and 2.12 of the online text of this Recommendation at http://www.w3.org/TR/2000/REC-xml-20001006

XML allows to add a keyword to the end of an attribute specification to make clear what action an XML processor should take when the attribute does not appear in a particular start tag. There are three possible keywords in this regard:

- **#REQUIRED** – this means the attribute is required for all instances of that particular start tag. Whenever the attribute is missing, the XML document will be invalid.

```
<!ATTLIST Student StudentID ID #REQUIRED>
```

- **#IMPLIED** – this keyword is used to make the XML processor tell the application that an attribute is missing in an instance of a start tag. It's than up to the application to decide what to do with it. Typically, this keyword is used when the application can construct (calculate) the value itself, e.g. by counting.

```
<!ATTLIST Student number.of.lessons #IMPLIED>
```

- **#FIXED** – this keyword is used to fix the value of a particular attribute. The keyword must precede the fixed value. Any value that is specified must match the fixed value, otherwise the document will be invalid.

```
<!ATTLIST Student membercode #FIXED "S">
```

The discussion whether to store data in the content model of an element or to save it as an attribute of an element is interesting. However, the general principle seems to be that metadata (information about information) should be stored in attributes and that the actual information should be saved in the content model.

2.4.3 XML Entities

Basically, XML entities are XML documents or data packets. Entities are used within attribute declarations for reuse of common constructions. They can be referenced using entity references. These references consist of a legal XML name (cfr. section 2.6), preceded by an ampersand (&) and followed by a semi-colon character (;). In general, XML makes a distinction between three types of entities:

- **General entities**

The simplest form of an entity is a general entity. It allows us to link a name to a bag of parsed character data:

```
<!ENTITY author "Kurt Lommens">
```

Every time the string *&author*, which is in fact an entity reference, appears in the XML document, the XML processor will replace it by *Kurt Lommens*. The use of the " character is obligatory both at the beginning and at the end of the replacement text and serves as delimiter. Note that the replacement text may contain marked up text.

- **Predefined entities**

In XML, any character may be referenced by a numeric reference. This is done by writing the characters *&#*, immediately followed by the numeric value of the character and a semi-colon at the end. For example, the numeric reference for the copyright symbol (©) would be *&#169* in decimal and *&#xA9* in hexadecimal.

Five characters are so commonly used that XML provides some predefined entities for them:

Character	Entity reference	Usage
&	&amp;	Always used to escape the & character (except within CDATA sections).
<	&lt;	Always used to escape the < character (except within CDATA sections).
>	&gt;	May be be used to escape the > character – within a CDATA section, the entity reference must always be used if the > follows a "]]" string literal.
‘	&apos;	May be used to escape the ‘ character in string literals.
“	&quot;	May be used to escape the “ character in string literals.

Table 2.5: XML predefined entities and their usage

- **Parameter entities**

Parsed entities that are used solely within the DTD are called parameter entities. They allows us easy reference of common constructs in the DTD. They are declared with the *ENTITY* keyword, a percent sign, a name and a replacement text. For example:

```
<!ENTITY % MemberParameters “Email ID #REQUIRED  name CDATA #REQUIRED”>
```

They function much the same as general entities, except for the fact that they are only allowed within DTD's and must be dereferenced using the percent sign, in casu by *%MemberParameters*, instead of the ampersand.

Except for the five predefined entities, all entities must be declared prior to their use in an XML document. Entity declarations can be both internal and external to the XML document they are referenced in.

Internal entity declarations must be defined within the XML document itself, using a section of the *internal subset*. All of the above examples are internal entity declarations.

External entity declarations must be defined in an *external subset*. The location of the content is identified using an external identifier. Usually, this identifier consists of the keyword *SYSTEM*, followed by a URI (cfr. higher):

```
<!ENTITY external.members SYSTEM “http://www.members.com/2001.xml”>
```

Using *&external.members* in an XML document linked to the DTD that holds this external entity declaration, will cause the reference to be replaced with the content of the file found at the specified internet address. Since this file is an XML document, its content will be XML parsable. XML also allows XML unparsable documents, such as graphic files, to be referenced:

```
<!ENTITY external.graphs SYSTEM "http://www.members.com/2001.gif" NDATA GIF>
```

Clearly, the *&external.graphs* entity reference does not point to a valid XML document. The keyword NDATA (Notation Data) and the explicit declaration of the notition name, in casu *GIF*, are obligatory in this case. Note that the use of the *NDATA* keyword is also allowed for internal entity declarations.

The power of XML entities and entity references consists of the fact that it makes possible to break up our XML documents into several pieces, allowing us to spread them all over different physical locations, such as the hard-disk, the network or even the internet. One can easily think of XML entity references as some kind of XML macro's.

2.5 XML and Style: Stylesheet Languages

So far, we have been talking mainly about data and structure in XML. Style, which is so prevalent in HTML, is not an integral part of the XML 1.0 Recommendation. In fact, we have by no means expressed or defined yet how our structured data should be displayed on a web page. The key is that stylistic rules should be stored seperately in a stylesheet.

The W3C offers two stylesheet languages to work with XML: Cascading Style Sheets (CSS1[14] and CSS2[15]) and the eXtensible Stylesheet Language[16] (XSL). Since the details of both languages go way beyond the scope of this thesis, they will not be explained any further.

2.6 XML naming rules

The XML 1.0 Recommendation specifies a few naming rules:

- *A name consists of at least one letter: a to z, or A to Z.*
- *If the name consists of more than one character, it may start with an underscore (_) or a colon (:). In actual practice, the colon character should only be used as a namespace delimiter (cfr. lower).*
- *The initial letter/underscore/colon can be followed by one or more letters, digits, hyphens, underscores, full stops and combining characters, extender characters and ignorable characters.*

By default, XML is based upon the Universal Character Set (UCS), defined in the *ISO/IEC 10646 UTF-8 character set standard*, which is currently congruent with the somewhat better-known *Unicode* standard. Nevertheless, XML enables programmers to assign specific character sets to declared sections of XML documents. This is major difference with HTML, which is based upon the *ISO 8859/1 character set*. This character set is basically an extension of the *ISO 646 character set*, better-known as *ASCII*.

[14] See http://www.w3.org/TR/REC-CSS1 for more information
[15] See http://www.w3.org/TR/REC-CSS2/ for more information
[16] See http://www.w3.org/TR/xsl/ for more information

2.7 XML related work at the W3C

2.7.1 The W3C Document Object Model (DOM) Working Group

Since the *XML 1.0 Specification* only provides the syntax and the grammar for XML, an Application Programming Interface (API) will be needed in order to be able to access the XML content in an application. This is where the work of the W3C Document Object Model (DOM) Working Group fits in. The DOM is a generic tree-based, platform- and language-neutral API that allows programs and scripts to access and update content, structure and style in a standard way. It has been designed with both HTML and XML in mind.

As for XML, using the DOM to access documents can be done in three ways:

- Using a scripting language (e.g. JavaScript, VBScript) in the document
- Using an external application such as a plug-in, or ActiveX control, that accesses the document trough the browser
- Using an external XML parser that implements the DOM.

The W3C DOM consists of different levels. Each new level is based on the previous one, extending the API to a more general use. This set of DOM levels is usually referred to as the DOM Specifications[17]**:**

- **Level 0:** Functionality equivalent to that exposed in Netscape Navigator 3.0 and Microsoft Internet Explorer 3.0. The W3C DOM builds on this technology and therefore offers no specification for this level.

- **Level 1:** This Level provides support for XML 1.0 and HTML. It was adopted as a W3C Recommendation (1 October, 1998), called *Document Object Model (DOM) Level 1 Specification - Version 1.0*. A more recent version (*Second Edition*) has been adopted as a W3C Working Draft (29 September, 2000). Both versions contain functionality for document navigation and manipulation.

- **Level 2:** This Level extends Level 1 with support for XML 1.0 namespaces and adds support for CSS (cfr. section 2.5), events (user interface and tree manipulation events) and enhances tree manipulations (tree ranges and traversal mechanisms). Most parts of this Level were adopted as W3C Recommendations (13 November, 2000). The HTML part of this Level is still in progress and has been adopted as a W3C Working Draft (13 November, 2000).

- **Level 3:** The third Level of DOM specifications is still in development. Level 3 will extend Level 2 by finishing support for XML 1.0 with namespaces and will extend the user interface events (keyboard, device independent events). It will also add abstract content model support, the ability to load and save a document or a content model, explore further mixed markup vocabularies and the implications on the DOM API ("Embedded DOM"), and will support XPath. In this regard, the W3C has adopted 3 Working Drafts (Core, Content models and Load and Save, Events) in 2001. The DOM Working Group plans to deliver a Recommendation for DOM Level 3 at the end of the year 2001. The *Views and Formatting Model* has been removed from Level 3 and is now presented as a public Working Draft. The W3C DOM Working Group waits for more experience and experimentation before going further on this model.

[17] See http://www.w3.org/DOM/DOMTR for more information

2.7.2 The W3C XML Schema Working Group

2.7.2.1 XML and syntactic complexity

XML DTD's are based on another syntax than XML, namely the Extended Backus Naur Form (EBNF). While EBNF is not a very efficient way to represent syntax for human understanding, it is a very solid way to represent the syntax for a language that will be parsed by a computer. Due to his complex syntax, writing and understanding valid XML DTD's is mostly a tough job.

This problem has been recognized by a lot of academic researchers. In recent years, there have been a number of proposals for alternatives to DTD's. The *XML Schema Working Group* at W3C has considered proposals like XML-Data[18], Document Content Description[19] (DCD), Schema for Object-Oriented XML[20] (SOX) and Document Definition Markup Language[21] (DDML, previously known as XSchema). All these proposals acknowledge the need for more rigorous and comprehensive facilities for declaring constraints on the use of markup[22]. The *XML Schema Working Group* has published a Requirements Document in early 1999 and the Specification (Primer, Structures, Datatypes) is a Candidate Recommendation as of October 2000.

2.7.2.2 XML and datatyping

Next to this problem of syntactic complexity, XML also suffers from a lack of support for datatyping. All XML documents are written with a single data type, i.e. text. XML does not provide a standard mechanism for including the non-textual type of data we want to markup. Working with DTD's implies that datatype restrictions are included and checked in external application software.

The lack of support for datatypes in XML has been dealt with by the *XML Schema Working Group* in their Specification document (Datatypes part). Appendix 2 lists all built-in datatypes and the values of their fundamental facets provided in this document. The document also provides an exhaustive list of constraining facets with all primitive and derived datatypes they apply to. The following constraining facets are defined: *length*, *minLength*, *maxLength*, *pattern*, *enumeration*, *maxInclusive*, *maxExclusive*, *minExclusive*, *minExclusive*, *precision*, *scale*, *encoding*, *duration* and *period*. This strong support for datatyping in XML Schema is an important improvement compared to XML.

2.7.2.3 XML and namespaces

XML does not provide a means to refer to an element definition in a DTD. We can refer to multiple DTD's in the same XML document, but this raises the problem of ambiguity and name collisions. Different DTD's can define XML elements with the same name and with a complete different structure. This constraint on the use of DTD's has a important negative influence on the flexibility of XML documents.

---

[18] See http://www.w3.org/TR/1998/NOTE-XML-data-0105/Overview.html for more information
[19] See http://www.w3.org/TR/NOTE-dcd for more information
[20] See http://www.w3.org/TR/NOTE-SOX/ for more information
[21] See http://www.w3.org/TR/NOTE-ddml for more information
[22] See http://www.w3.org/XML/Activity.html for more information

This problem has been addressed by the W3C in a Recommendation called *Namespaces in XML*[23] (14 January 1999). In this Recommendation, namespaces are introduced as a resource from which we can take whatever definition we need. This idea of namespaces has been adopted by the *XML Schema Working Group* in their specification document (Primer part).

2.7.2.4 XML and inheritance

With DTD's, there is no good way to express inheritance. We will try to find a solution for this problem when elaborating our mapping algorithm (cfr. section 3) but this solution will only be artificial. No syntax is provided by XML to define XML elements that inherit all properties from other XML elements in a straightforward way. This problem has only been partially solved in XML Schema. While XML Schema supports a single inheritance model, multiple inheritance remains a problem field. Next to this, XML Schema does not support exclusion or subset constraints that target non-key elements.

2.7.2.5 XML and set constraints

XML supports set constraints on an attribute by allowing an enumerated list of values for that attribute. No syntax is provided to describe large data sets other than defining all possible values in this enumerated list. This shortcoming in XML is dealt with in XML Schema by the introduction of boundaries to describe (logical) sequences of values.

In addition, the XML occurrence operators do not suffice to restrict the occurrence of XML elements in XML documents in a flexible way. The only provided boundaries are zero, one and infinite. XML does not support occurrence restrictions higher than one and lower than infinite. This large gap is closed in XML Schema by the introduction of attributes *minOccurs* and *maxOccurs*. The value of these attributes is always of type *integer*, except when *maxOccurs* is intended to be infinite[24].

2.7.2.6 XML and identifier constraints

No XML element can have more than one attribute of type ID. In many cases, XML elements hold more than one unique attribute. In order to ensure the uniqueness of attributes that are not of type ID, external integrity checking is needed on declared elements in XML documents. Moreover, unique attributes that are not of type ID can never be used to refer to the XML element holding the attribute. To our knowledge, this problem has not been addressed yet by the W3C.

[23] See http://www.w3.org/TR/1999/REC-xml-names-19990114/ for more information
[24] In that case, the value for 'maxOccurs' is marked as '*' or 'unbound'

2.7.3 The W3C XML Linking Working Group

One important feature of HTML is the ability to insert hypertext links in websites. In this regard, the WWW can be seen as a set of related sites. The XML Linking Working Group is designing hypertext links for XML. Their objective is to design advanced, scalable, and maintainable hyperlinking and addressing functionality for XML. They make a clear distinction for links between objects - *external links*, and links to locations within XML documents - *internal links.*

2.7.3.1 XML Linking Language (XLink)[25]

XLink is designed to deal with *external links* in XML. It has been adopted by the W3C as a Proposed Recommendation (20 December, 2000). It allows elements to be inserted into XML documents in order to create and describe links between resources. It uses XML syntax to create structures that can describe links similar to the simple unidirectional hyperlinks of today's HTML, as well as more sophisticated links.

Basically, XLink allows documents to:

- Assert linking relationships among more than two resources
- Associate metadata with a link
- Express links that reside in a location separate from the linked resources

The model defined in the XLink Specification shares with HTML the use of URI technology, but goes beyond HTML in offering features, previously available only in dedicated hypermedia systems, that make hyperlinking more scalable and flexible.

2.7.3.2 XML Base[26]

One of the stated requirements on XLink is to support HTML linking constructs in a generic way. The *HTML BASE* element is one such construct which the XLink Working Group has considered. *BASE* allows authors to explicitly specify a document's base URI for the purpose of resolving relative URI's in links to external images, applets, form-processing programs, style sheets, and so on. The syntax consists of a single XML attribute name *xml:base.*

XML Base has also been adopted by the W3C as a Proposed Recommendation (20 December, 2000). Both XLink and XML Base are currently awaiting a decision after the Advisory Committee review.

2.7.3.3 XML Pointer Language (XPointer)[27]

XPointer, which is based on the XML Path Language[28] (XPath), supports addressing into the internal structures of XML documents. It allows for traversals of a document tree and choice of its internal parts based on various properties, such as element types, attribute

[25] See http://www.w3.org/TR/xlink/ for more information
[26] See http://www.w3.org/TR/xmlbase/ for more information
[27] See http://www.w3.org/TR/WD-xptr for more information
[28] See http://www.w3.org/TR/xpath for more information

values, character content, and relative position. It supports addressing into the internal structures of XML documents. It allows for examination of a document's hierarchical structure and choice of its internal parts based on various properties, such as element types, attribute values, character content, and relative position. In particular, it provides for specific reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute.

XPointer has been adopted by the W3C as a Last Call Working Draft (8 January, 2001).

2.7.4 The W3C XML Query Working Group

The work of this Working Group is situated in the field of data extraction, exchange and integration for XML documents. The existing query languages, either relational (Structured Query Language - SQL) or object-oriented (OQL), do not immediately apply to XML because they are only targeted at particular data type sources. Therefore, the XML Query Language[29] (XQuery) has been designed to be broadly applicable across all types of XML data sources.

The XML Query Working Group published a revised set of documents in February 2001. These documents include the following Working Drafts:

- XML Query Requirements
- XML Query Use Cases
- XML Query Data Model
- The XML Query Algebra
- XQuery: A Query Language for XML

At the same time the XML Query Working Group co-authored a requirements document for XPath 2.0 with the XSL Working Group.

2.7.5 Other W3C XML Working Groups

2.7.5.1 The W3C XML Core Working Group

The mission of the XML Core Working Group is to elaborate the XML 1.0 Recommendation, maintaining it in response to reported errata and other comments, and providing essential supplementary materials.

2.7.5.2 The W3C XML Coordination Working Group

The membership of this group is the chairs of the individual Working Groups. Its role is to provide a forum for coordination between the Working Groups of the XML Activity, and between the XML Activity and other parts of W3C, and between the XML Activity and other organizations.

[29] See http://www.w3.org/TR/xquery/ for more information

2.7.5.3 The W3C XML Signature Working Group[30]

Nowadays software systems often use digital signatures in order to validate specific requests. In this regard, we can think of home-banking transactions, employees trying to access restricted data, etc. In view of the latest developments on the WWW, the XML Signature Working Group is working on the digital signing of XML. This capability is critical for a variety of electronic commerce applications, including payment tools.

2.7.5.4 The W3C XML Encryption Working Group[31]

The mission of this Working Group is to develop a process for encrypting/decrypting digital content (including XML documents and portions thereof) and an XML syntax used to represent the encrypted content and information that enables an intended recipient to decrypt it.

2.8 XML and Conceptual Data Modeling

In section 2.4 of this document, we have introduced DTD's as the mechanism for defining structure of data instances in XML documents. The XML 1.0 Recommendation provides a clear syntax for valid DTD construction. However, it does not provide a view or a technique to model DTD's at the conceptual level.

In the next section, we will try to use ORM as a conceptual data modeling technique in the design process of DTD's. In order to do so, we will elaborate an algorithm that produces a DTD from an ORM CS. The basic ideas of this algorithm can be summarized like this:

- Map NOLOT's to XML elements and use their primary reference as an attribute of type *ID*.
- Use the content model of XML elements mapped from NOLOT's, combined with the XML order and cardinality operators, to express inheritance.
- Map LOT's that are not involved in non-explicit primary references to XML elements of type *#PCDATA*.
- For unnested fact types, map roles played by NOLOT's to empty XML elements. Express relationships by the use of attributes of type IDREF and/or the content model of the mapped XML elements.
- Nested fact types should be considered and mapped as artificial NOLOT's
- Identifier and mandatory constraints should be mapped using the XML cardinality operators in the content model of mapped XML elements.

[30] see http://www.w3.org/Signature/ for more information
[31] see http://www.w3.org/Encryption/2001/ for more information

3. Using ORM CS's to build XML DTD's

3.1 Assumptions

This section describes an algorithm to map a maximum of syntactic and semantic properties from an ORM CS to an XML DTD. All steps in the algorithm will be illustrated using the sample CS constructed throughout section 1 of this document.

For reasons of simplification, we assume CS's to hold only binary fact types. Of course, methods can be developed to map (sets of) fact types between two or more object types, but this goes beyond the scope of this text.

Next to this, all CS's to be mapped are assumed to be valid. This assumption might seem trivial at first sight. Looking closer at rather complicated conceptual models, one might find out that validation is sometimes a complex matter. Even small mistakes like unreferenced NOLOT's and misplaced identifier constraints can have a large effect on the output of this algorithm. Therefore, the use of a validating tool on all CS's is recommended[32].

3.2 Non-subtype NOLOT's

RULE 1*: Determine a primary reference for each non-subtype NOLOT. Mark all involved Object Types as "used". For non-explicit references, also mark all involved roles as "used".*

When this rule is applied to our sample CS, the following table of primary references will be obtained:

Non-Lexical Object Type	**Primary reference**	
	Lexical Object Type	**Non-Lexical Object Type**
Member	Email	-
Course	Code	-
Day	Daynr	-
Month	Monthnr	-
Year	Yearnr	-
Date	-	{Day, Month, Year}

Table 3.1: Primary references for non-subtype NOLOT's

All object types present in table 3.1 must be marked as *used*. Since NOLOT *Date* has no explicit reference, the algorithm will find the combination of NOLOT's *Day*, *Month* and *Year* to be a reference for *Date*. Following Rule 1, the roles between *Date* and *Day*, *Date* and *Month*, *Date* and *Year* must also be marked as *used*.

RULE 2: *Define each non-subtype NOLOT with a unary primary reference as an XML element. If the primary reference is lexical, use it to define a #REQUIRED attribute of type ID for this XML element. Else, use it to define a #REQUIRED attribute of type IDREF for this XML element.*

[32] e.g. the validating tool provided by VisioModeler.

Since this rule does not mention the content model of the newly declared XML elements, only incomplete code fragments will be produced. For a better understanding, we have placed "XXX" as a temporarily content model[33]:

```
<!ELEMENT Member XXX>
<!ATTLIST Member email ID #REQUIRED>

<!ELEMENT Course XXX>
<!ATTLIST Course code ID #REQUIRED>

<!ELEMENT Day XXX>
<!ATTLIST Day daynr ID #REQUIRED>

<!ELEMENT Month XXX>
<!ATTLIST Month monthnr ID #REQUIRED>

<!ELEMENT Year XXX>
<!ATTLIST Year yearnr ID #REQUIRED>
```

Our CS does not contain any NOLOT's having another NOLOT as their unary primary reference. However, we can easily demonstrate the working of Rule 2 in such cases by widening our sample CS a little bit. Suppose each Member holds a MemberCard, which has a StartDate and an EndDate. Our sample CS could be changed like this (*StartDate* and *EndDate* are modeled to be LOT's for reasons of brevity):

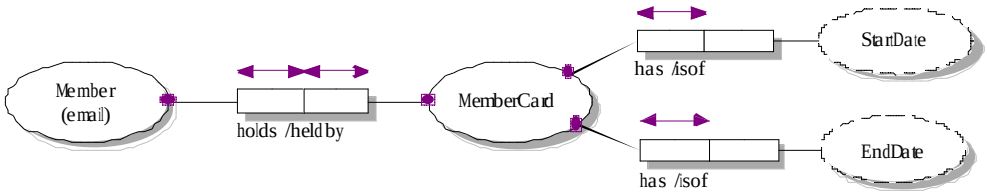


Figure 3.1: sample ORM diagram holding a NOLOT with another NOLOT as its unary primary reference

In the above diagram, NOLOT *MemberCard* can only be referenced by NOLOT *Member*. Following Rule 2, code fragments like these would be generated:

```
<!ELEMENT Member XXX>
<!ATTLIST Member email ID #REQUIRED>

<!ELEMENT MemberCard XXX>
<!ATTLIST MemberCard Member IDREF #REQUIRED>
```

Using an attribute of type IDREF in XML defines a link between the element holding the attribute and another element. This other element has to be declared before the element referring to it can be used. In our case, an instance of the element *Member* has to be declared before we can declare an instance of the element *MemberCard* referring to the instance of element *Member*. In any other case, the XML processor will call the XML document to be invalid.

[33] This will be addressed in section 3.4 and 3.6.

The problem with attributes of type IDREF is that XML offers no means to target the link to a specific element type. The XML processor just checks whether an instance of ANY element has been declared holding an attribute of type ID with the same value as the value of the attribute of type IDREF, even if the name of this attribute equals the name of the targeted element. Suppose the XML processor finds an instance of element *MemberCard* in an XML document, holding *david.williams@vub.ac.be* as the value of the attribute *Student*. In this case, it will suffice for the processor to find an earlier declaration of an instance of any element holding *david.williams@vub.ac.be* as the value of its type ID attribute, not to generate an error.

In order to assure the correct linking between an attribute of type IDREF and its target element, we need to include an external integrity check. Such checks are commonly used when mapping CS's to relational database structures. Since there is no algorithm that maps the complete semantics of an ORM schema to a relational database structure, most constraints drawn on CS's are exported and included in application software. This software acts as a filter for all data to be inserted in the actual tables of the database. Of course, this solution is artificial and makes it more difficult to understand the generated XML DTD.

RULE 3: *Define each non-subtype NOLOT with a non-unary primary reference as an XML element with all parts of the primary reference in its content model. Use a dummy name to define a #REQUIRED attribute of type ID for this XML element.*

Looking at Table 3.1, *Date* is the only NOLOT meeting the requirements of this rule. The following code fragment will be produced:

```
<!ELEMENT Date (Day, Month, Year)>
<!ATTLIST Date dummy_date ID #REQUIRED>
```

This construction reveals another important shortcoming in the XML syntax: XML does not support attributes of type ID that reflect a non-unary primary reference. We have solved this by the introduction of a dummy ID attribute holding a unique value combined with an external integrity check on the combinations of the identifying values provided by XML elements *Day*, *Month* and *Year*. Solving the problem this way makes it possible though to use the dummy_day attribute as an attribute of type IDREF in the attribute list of other defined XML elements. Since attributes of type ID and IDREF act respectively like primary keys and foreign keys in relational databases, the importance of this shortcoming becomes clear. A similar problem occurs for NOLOT's with multiple (unique) references. This will be discussed in section 3.6.3.1 (part e).

3.3 Subtype NOLOT's

RULE 4: *If possible, determine a primary reference for each subtype NOLOT. Mark all involved Object Types as "used". For non-explicit references, also mark all involved roles as "used". If no primary reference is found, use the primary reference of the supertype NOLOT as the primary reference for the subtype NOLOT.*

When this rule is applied to our sample CS, the following table of primary references will be obtained:

Non-Lexical Object Type	Primary reference	
	Lexical Object Type	Non-Lexical Object Type
Professor	ProfessorID	-
Student	StudentID	-

Table 3.2: Primary references for subtype NOLOT's

All object types appearing in Table 3.2 must be marked as being used. Since both *professorID* and *studentID* are non-explicit references, the roles between *Professor* and *ProfessorID*, *Student* and *studentID* must also be marked as being used.

RULE 5: *Define each subtype NOLOT with a unary primary reference as an XML element. If the primary reference is lexical, use it to define a #REQUIRED attribute of type ID for this XML element. Else, use it to define a #REQUIRED attribute of type IDREF for this XML element.*

Rule 5 will produce the following code fragments:

```
<!ELEMENT Professor XXX>
<!ATTLIST Professor professorID ID #REQUIRED>

<!ELEMENT Student XXX>
<!ATTLIST Student studentID ID #REQUIRED>
```

When a subtype NOLOT has another NOLOT as its primary reference, we must port the implicit link to application software. Using an attribute of type IDREF only partially solves the problems, as discussed in section 3.2 (cfr. Rule 2).

RULE 6: *Define each subtype NOLOT with a non-unary primary reference as an XML element with all parts of the primary reference in its content model. Use a dummy name to define a #REQUIRED attribute of type ID for this XML element.*

Rule 6 defines subtype NOLOT's with a non-unary primary reference exactly the same way as described in Rule 3. Since no subtype NOLOT on our sample CS meets the requirements of this Rule, no code fragments are generated.

The XML 1.0 Recommendation does not allow elements to hold more than one attribute of type ID. Therefore, our solution for subtype NOLOT's will not work for multiple inheritance models (cfr. also section 4.1.4).

3.4 Linking subtype and supertype NOLOT's

So far, the generated code fragments for subtype NOLOT's lack the necessary link(s) to their supertype(s). Subtypes do not only inherit all properties of supertypes, they are also subject to exclusion and totality constraints with regard to the involved supertypes. Since the XML syntax does not support inheritance models (cfr. section 4.1.4), only an artificial solution can be found to express the relationship between subtypes and supertypes.

RULE 7: *For each supertype NOLOT, build a list of its subtypes. Depending on the exclusion and totality constraints, structure this list as follows:*

a. *Total and Exclusive:*
   Use the '|' separator between all subtypes in the list.

b. *Not Total and Exclusive:*
   Add the '?' cardinality operator after all subtypes in the list and use the '|' separator between all subtypes in the list.

c. *Total and non-Exclusive:*
   If a supertype has n different subtypes, build all possible distinct lists of n different subtypes with (n-1) subtypes holding the '?' cardinality operator. Make a '|'-separated list of all constructed lists.

d. *Not Total and non-Exclusive:*
   Add the '?' cardinality operator after all subtypes in the list and use the ',' separator between two or more subtypes in the list.

Add the constructed list to the content model of the involved supertypes.

In our sample CS, we find one supertype NOLOT (Member) linked to two subtype NOLOT's (Professor and Student). Since every Member must be either a Professor or a Student, a totality and exclusion constraint must be placed between both subtypes. This is clearly an example of case a in Rule 7. Hence, our earlier declaration of the XML element Member will be completed like this:

```
<!ELEMENT Member (Professor | Student)>
<!ATTLIST Member email ID #REQUIRED>
```

Dropping the totality constraint (case b) would mean that every Member could be either a Professor or a Student. It is also possible that a Member is none of both:

```
<!ELEMENT Member (Professor? | Student?)>
<!ATTLIST Member email ID #REQUIRED>
```

Dropping the exclusion constraint (case c) would mean that every Member must be at least a Professor or a Student. It is also possible that a Member is both:

```
<!ELEMENT Member ((Professor , Student?) | (Professor? , Student))>
<!ATTLIST Member email ID #REQUIRED>
```

Dropping both constraints (case d) would mean that every Member could be either a Professor or a Student. A Member could also be both and a Member could also be none of both:

```
<!ELEMENT Member (Professor? , Student?)>
<!ATTLIST Member email ID #REQUIRED>
```

In the programming world, inheritance models ensure that a subtype inherits all properties from its supertype(s). In other words, all properties of a supertype are linked to its subtype(s). Our solution turns this idea of inheritance somehow upside down. In fact, we link all properties of a subtype to its supertype(s) by adding subtype elements to the content model of the supertype element(s). Working this way makes it possible to express exclusion and totality constraints in the content model of supertype elements, as shown in Rule 7. However, this technique has an important shortcoming: The definition of a subtype element does not express the link to its supertype(s). External integrity checking will be needed based upon the exclusion and totality constraints to ensure the validity of XML instances of both subtype and supertype elements. Why do we work this way ?

Following the basic idea of inheritance, we would have to add supertype elements to the content model of subtypes. Looking at our sample CS, code fragments like this would be generated:

```
<!ELEMENT Member XXX >
<!ATTLIST Member email ID #REQUIRED>

<!ELEMENT Professor (Member)>
<!ATTLIST Professor professorID ID #REQUIRED>

<!ELEMENT Student (Member)>
<!ATTLIST Student studentID ID #REQUIRED>
```

This technique respects the basic idea of inheritance. However, it does not allow exclusion or mandatory constraints to be mapped to the generated XML DTD. Therefore, both constraints must be ported to an external integrity checking tool. This is the main reason why we have chosen to map inheritance models as described in Rule 7.

A lot of academic research work has been done on this important shortcoming in the XML syntax. In section 4, the XML Schema language will be presented as an extension to XML. This new syntax partially supports inheritance models by allowing single inheritance. Multiple inheritance is still not supported.

3.5 LOT's

So far, all NOLOT's, all LOT's and roles involved in non-explicit references and all explicit references are marked as being *used*. What about non-marked LOT's ?

RULE 8: *Define each non-marked LOT as an XML element of type #PCDATA. Mark all involved LOT's as being "used".*

In our sample case, only *membercode* and *name* are left as non-marked LOT's, resulting in the following code fragments after applying Rule 8:

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

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

The above code fragments reveal another important shortcoming in the XML syntax. XML provides a very poor support for datatyping. There is no possible way to specify in our XML DTD that the value held by element *membercode* should be of type *char* and the value held by element name should be of type S*tring*. We will deal with this problem in section 4.1.3.

Apart from the datatype, our definition of XML element *membercode* does not reflect its restricted content (set constraint). Looking at our sample CS, we can see that a *membercode* element can only hold two possible values, namely *P* and *S*. Once again, the XML syntax provides a very poor support for restricted data sets (see section 4.1.5). Of course, we can use an enumerated attribute. By doing so, we would have to define *membercode* as an EMPTY element, holding one dummy enumerated attribute:

```
<!ELEMENT membercode EMPTY>
<!ATTLIST membercode dummy_code (P | S) #REQUIRED>
```

In our example, it would be a fair solution. But what about large data sets ? ORM allows value constraints like {1..100}, indicating integers ranging from 1 to 100. On the contrary, the XML syntax does not support value ranges, it simply forces us to define each and every possible value. Naturally, this is very impractical.

Being where we are now, our generated code fragments do not (yet) reflect the textual constraints of both subtypes on our CS. By defining the XML element *membercode* without an attribute of type ID, we can not refer to it. Mapping the internal definitions of our subtypes can be done in two distinct ways, both leaving out the definition of *membercode* as an XML element:

The first way is to define *membercode* as an attribute with a #FIXED (distinct) default value for both a *Professor* and a *Student*. In the case of a *Professor*, a code fragment like this could be generated:

```
<!ELEMENT Professor XXX>
<!ATTLIST Professor
        professorID ID #REQUIRED
        membercode “P” #FIXED>
```

Although this solution seems intuitively correct, it has a small shortcoming. Most XML parsers do not generate an error if a #FIXED attribute (with a default value) does not appear. They simply assume it to have the default value[34]. This could lead to validated instances of the XML element *Professor* that lack the membercode attribute and thereby suffer data loss.

The second, perhaps better way is to define membercode as a #REQUIRED enumerated attribute, allowing the group of possible values to hold only one value. In the case of a *Professor*, a code fragment like this could be generated:

```
<!ELEMENT Professor XXX>
<!ATTLIST Professor
        professorID ID #REQUIRED
        membercode (P) #REQUIRED>
```

[34] Didier MARTIN, Mark BIRBECK, Michael KAY, e.a., 2000, Professional XML, Wrox Press Ltd., Burmingham, UK, pg 83

With this solution, all XML parsers will be forced to check the membercode attribute. Instances without or with a false *membercode* attribute will always cause an XML parser to generate an error.

The textual constraints shown on our sample CS are based on RIDL definitions. Both presented solutions to map these constraints from a CS to an XML DTD are mainly based on human understanding of the UoD. Therefore, they are not presented as a part of our mapping algorithm. External integrity checks will be needed to ensure the correctness of instances based on our XML DTD.

3.6 Fact types

3.6.1 Unnested fact types

Now that we have used all object types from our sample CS to define the necessary XML elements and attributes, we still need a way to include most of the roles played by those object types in our XML DTD. Clearly, we will have to make a distinction between nested and unnested Fact Types.

RULE 9: *For unnested fact types between NOLOT's, define all non-marked roles as EMPTY XML elements. Use the (ID reference of the) co-NOLOT to add a #REQUIRED attribute of type IDREF to this XML element.*

This rule assumes an attribute of type ID to be defined for all NOLOT's. This means we are obliged to use a dummy attribute of type ID for NOLOT's with a non-unary primary reference (cfr. Rule 3). Applying Rule 9 to our sample CS will produce the following code fragments:

```
<-- unnamed Fact Type -->
<-- NOLOT Member and NOLOT Date -->

<!ELEMENT born_on EMPTY>
<!ATTLIST born_on dummy_date IDREF #REQUIRED>
<!ELEMENT birthdate_of EMPTY>
<!ATTLIST birthdate_of email IDREF #REQUIRED>

<-- unnamed Fact Type -->
<-- NOLOT Professor and NOLOT Course -->

<!ELEMENT teaches EMPTY>
<!ATTLIST teaches code IDREF #REQUIRED>
<!ELEMENT teached_by EMPTY>
<!ATTLIST teached_by professorID IDREF #REQUIRED>

<-- unnamed Fact Type -->
<-- NOLOT Student and NOLOT Course -->

<!ELEMENT follows EMPTY>
<!ATTLIST follows code IDREF #REQUIRED>
<!ELEMENT followed_by EMPTY>
<!ATTLIST followed_by studentID IDREF #REQUIRED>
```

Comment sections stating the name (if present) and the players involved in each fact type, are included in the code fragments. This ensures a better understanding of the generated XML DTD.

RULE 10: *For unnested fact types between a NOLOT and a LOT, define the non-marked role played by the NOLOT as an XML element with the name of the involved LOT in its content model.*

In contradiction to fact types between NOLOT's, we do not (completely) map the binary relationship for fact types between a NOLOT and a LOT. As described in section 3.5, a LOT is always considered to be a terminal leave and therefore lacks an attribute of type ID. For this reason, we can not map fact types between a NOLOT and a LOT the same way fact types between NOLOT's are mapped (cfr. Rule 9). Applying Rule 10 to our sample CS will produce the following code fragments:

```
<-- unnamed Fact Type -->
<-- NOLOT Member and LOT membercode -->

<!ELEMENT holds (membercode)>

<-- unnamed Fact Type -->
<-- NOLOT Member and LOT name -->

<!ELEMENT has (name)>
```

Another solution would be to define the non-marked role played by the NOLOT as an EMPTY XML element, holding the name of the involved LOT as an attribute of type CDATA:

```
<-- unnamed Fact Type -->
<-- NOLOT Member and LOT membercode -->

<!ELEMENT holds EMPTY>
<!ATTLIST holds membercode CDATA #REQUIRED>

<-- unnamed Fact Type -->
<-- NOLOT Member and LOT name -->

<!ELEMENT has EMPTY>
<!ATTLIST has name CDATA #REQUIRED>
```

Although this method seems attractive at first sight, it does not reflect the basic idea of ORM object types being mapped to XML elements and is therefore not used.

RULE 11 (together with Rule 9 and 10): *If a role in an unnested fact type is played by a NOLOT, add the rolename to the content model of the XML element defined earlier based on this NOLOT.*

In our sample case, Rule 11 will complete the content models of earlier defined XML elements like this:

```
<!ELEMENT Member ((Professor | Student), holds, has, born_on)>
<!ATTLIST Member email ID #REQUIRED>

<!ELEMENT Course (teached_by, followed_by)>
<!ATTLIST Course code ID #REQUIRED>

<!ELEMENT Date (daynr, monthnr, yearnr, birthdate_of)>
<!ATTLIST Date dummy_date ID #REQUIRED>

<!ELEMENT Professor (teaches)>
<!ATTLIST Professor professorID ID #REQUIRED>

<!ELEMENT Student (follows)>
<!ATTLIST Student studentID ID #REQUIRED>
```

3.6.2 Nested fact types

Now that all unnested fact types are mapped, we can deal with nested fact types. Since our sample CS does not include any nested fact types, we will extend a fact type from our CS to illustrate code fragments generated in this section.

Consider the fact type between NOLOT *Student* and NOLOT *Course*. Nesting of this fact type can be used to express that a Student following a Course can obtain a Score for that Course:

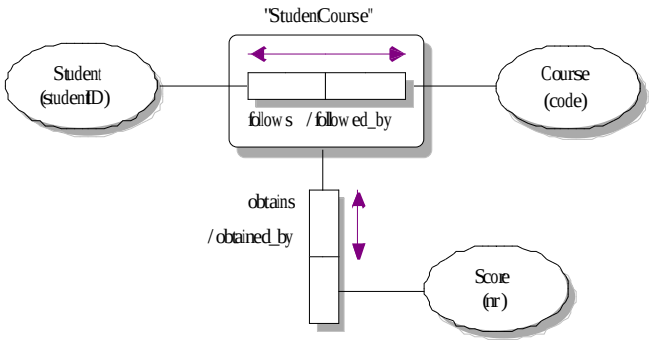


Figure 3.2: Nested fact type between NOLOT's

Applying the above Rules to this CS will generate the following code fragments:

```
<!ELEMENT Student (follows)>
<!ATTLIST Student studentID ID #REQUIRED>

<!ELEMENT Course (followed_by)>
<!ATTLIST Course code ID #REQUIRED>

<!ELEMENT Score XXX>
<!ATTLIST Score nr ID #REQUIRED>
```

```
<-- unnamed Fact Type -->
<-- NOLOT Student and NOLOT Course -->

<!ELEMENT follows EMPTY>
<!ATTLIST follows code IDREF #REQUIRED>
<!ELEMENT followed_by EMPTY>
<!ATTLIST followed_by studentID IDREF #REQUIRED>
```

RULE 12: *For nested fact types between NOLOT's, use the name of the nested fact type to define an XML element holding both involved rolenames in its content model. Use a dummy name to define a #REQUIRED attribute of type ID for this XML element. Consider this new XML element as (mapped from) an artificial NOLOT and use the Rules for unnested fact types to map all fact types this artificial NOLOT is involved in.*

Applying this rule to the sample CS in Figure 3.2 will generate the following code fragments:

```
<!ELEMENT StudentCourse (follows, followed_by, obtains)>
<!ATTLIST StudentCourse dummy_ID ID #REQUIRED>

<!ELEMENT Score (obtained_by)>
<!ATTLIST Score nr ID #REQUIRED>

<-- unnamed Fact Type -->
<-- NOLOT StudentCourse and NOLOT Score -->

<!ELEMENT obtains EMPTY>
<!ATTLIST obtains nr IDREF #REQUIRED>
<!ELEMENT obtained_by EMPTY>
<!ATTLIST obtained_by dummy_ID IDREF #REQUIRED>
```

Since nested fact types are treated as (artificial) NOLOT's, we have to introduce a dummy identifier for that NOLOT in order to make it possible to refer to the generated XML element. Once again, this is an artificial solution that makes generated XML DTD's less readable.

In section 3.6.1, we stated that roles played by LOT's in fact types between a NOLOT and a Lot are not mapped to XML elements. Therefore, we can not use Rule 12 to map nested fact types between a NOLOT and a LOT. This can be illustrated by using a slightly adapted version of the CS presented in Figure 3.2:

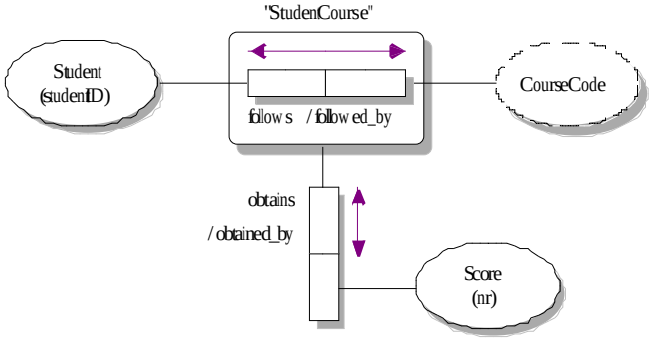


Figure 3.3: Nested fact type between a NOLOT and a LOT

Applying the above Rules to the CS presented in Figure 3.3 will generate the following code fragments:

```
<!ELEMENT Student (follows)>
<!ATTLIST Student studentID ID #REQUIRED>

<!ELEMENT Score XXX>
<!ATTLIST Score nr ID #REQUIRED>

<!ELEMENT CourseCode (#PCDATA)>

<-- unnamed Fact Type -->
<-- NOLOT Student and LOT CourseCode -->

<!ELEMENT follows EMPTY>
<!ATTLIST follows CourseCode CDATA #REQUIRED>
```

Because the role *followed_by* is not mapped to an XML element, an extra Rule for nested fact types between a NOLOT and a LOT has to be formulated:

RULE 13: *For nested fact types between a NOLOT and a LOT, use the name of the nested fact type to define an XML element holding the name of the role played by the NOLOT in its content model. Use a dummy name to define a #REQUIRED attribute of type ID for this XML element. Use (a reference to) the NOLOT to define a #REQUIRED attribute of type IDREF for this XML element. Consider this new XML element as (mapped from) an artificial NOLOT and use the Rules for unnested fact types to map all fact types this artificial NOLOT is involved in.*

With this Rule, we are able to map the nested fact type in Figure 3.3:

```
<!ELEMENT StudentCourse (follows, obtains)>
<!ATTLIST StudentCourse dummy_ID ID #REQUIRED
                        student_ID IDREF #REQUIRED>

<!ELEMENT Score (obtained_by)>
<!ATTLIST Score nr ID #REQUIRED>

<-- unnamed Fact Type -->
<-- NOLOT StudentCourse and NOLOT Score -->

<!ELEMENT obtains EMPTY>
<!ATTLIST obtains nr IDREF #REQUIRED>
<!ELEMENT obtained_by EMPTY>
<!ATTLIST obtained_by dummy_ID IDREF #REQUIRED>
```

3.6.3 Identifier and mandatory constraints

So far, we have been able to map unnested and nested fact types. What about identifier and mandatory constraints on fact types? Both constraints are static by definition, i.e. they apply at any time to each and every state of the relational database described by the CS. Mapping these constraints from a CS to an XML DTD can be (partly) done by using the XML cardinality operators.

RULE 14: *Add identifier and mandatory constraints using the XML cardinality operators. Mark all roles and nested fact types as being "used".*

3.6.3.1 Binary Fact Types between a NOLOT and a LOT

a. Mandatory and unique

Consider a UoD in which a Member is referenced by a unique memberID. Every Member must have exactly one name. Different Members can have the same name. The conceptual model of this UoD is presented as an ORM fact type in Figure 3.4:

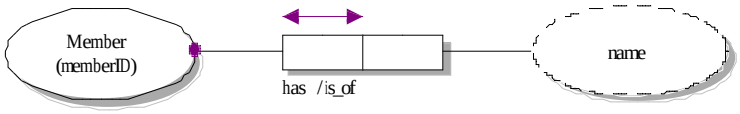


Figure 3.4: NOLOT-LOT fact type with a mandatory constraint and uniqueness on one role.

Mapping this ORM fact type by use of the above rules would result in the following (partial) XML DTD:

```
<!ELEMENT Member (has)>
<!ATTLIST Member memberID ID #REQUIRED>

<!ELEMENT has (name)>

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

The identifier and mandatory constraints are mapped to the DTD by using the non-visible cardinality operator in the content model of the XML element *Member*. As described earlier, the role *is_of* will not be mapped to an XML element. This will cause a problem when the role is marked with a uniqueness constraint. We will deal with this problem in part c of this section.

b. Non-mandatory and unique

Consider the UoD in part a of this section. Remove the mandatory constraint by allowing Members not to have a name. The conceptual model of this UoD is presented as an ORM fact type in Figure 3.5:

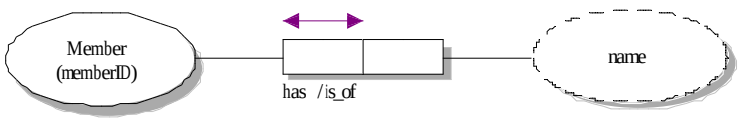


Figure 3.5: NOLOT-LOT fact type without a mandatory constraint and with uniqueness on one role.

Mapping this ORM fact type by use of the above rules would result in the following (partial) XML DTD:

```
<!ELEMENT Member (has)?>
<!ATTLIST Member memberID ID #REQUIRED>

<!ELEMENT has (name)>

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

The identifier and mandatory constraints are mapped to the DTD by using the '?' cardinality operator in the content model of the XML element *Member*.

c. Mandatory and non-unique

Consider a UoD in which a Member is referenced by a unique memberID. Every Member must have at least one sponsor. Every Member can have more than one sponsor. Different Members can have the same sponsor. The conceptual model of this UoD is presented as an ORM fact type in Figure 3.6:

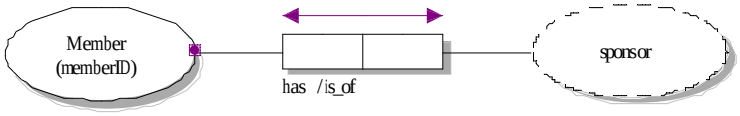


Figure 3.6: NOLOT-LOT fact type with a mandatory constraint and no uniqueness on a single role.

Mapping this ORM fact type by use of the above rules would result in the following (partial) XML DTD:

```
<!ELEMENT Member (has)+>
<!ATTLIST Member memberID ID #REQUIRED>

<!ELEMENT has (sponsor)>

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

The identifier and mandatory constraints are mapped to the DTD by using the '+' cardinality operator in the content model of the XML element *Member*.

d. Non-mandatory and non-unique

Consider the UoD in part c of this section. Remove the mandatory constraint by allowing Members not to have a sponsor. The conceptual model of this UoD is presented as an ORM fact type in Figure 3.7:

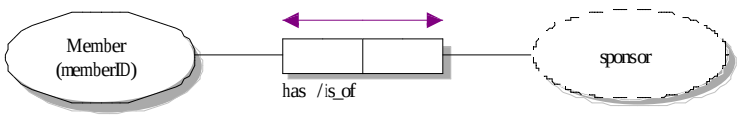


Figure 3.7: NOLOT-LOT fact type without a mandatory constraint and with no uniqueness on a single role.

Mapping this ORM fact type by use of the above rules would result in the following (partial) XML DTD:

```
<!ELEMENT Member (has)*>
<!ATTLIST Member memberID ID #REQUIRED>

<!ELEMENT has (sponsor)>

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

The identifier and mandatory constraints are mapped to the DTD by using the '*' cardinality operator in the content model of the XML element Member.

So far, all described constructions completely reflect the semantics of the constraints specified in the CS's. A problem occurs when a NOLOT has more than one identifying reference.

e. Problem Case: Multiple references for one NOLOT

Consider a UoD in which a Member is referenced by a unique memberID. Every Member must have exactly one worldranking. Different Members can not have the same worldranking. In this UoD, both memberID and worldranking can be used as an identifying reference for a Member. The conceptual model of this UoD is presented as an ORM fact type in Figure 3.8:

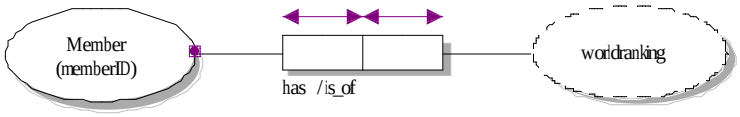


Figure 3.8: NOLOT-LOT fact type with a mandatory constraint and a uniqueness constraint on both roles.

The XML cardinality operators will not suffice to map the complete semantics of this conceptual model to a (partial) XML DTD. Depending on the (absence) of the mandatory constraint, we would either want to use the non-visible cardinality operator (cfr. part a in this section) or the '?' cardinality operator (cfr. part b in this section) in the content model of the XML element *Member*. In both cases, the uniqueness constraint on the role *is_of* is discarded since the role itself is not mapped to an XML element. This means that a validating parser on XML documents based on our DTD would not generate an error when two (or more) XML instances of *Member* have exactly the same worldranking.

Intuitively, one could argue that this problem can easily be solved by adding a new attribute *worldranking* of type ID to the attribute list of element *Member*. However, this approach would create an incorrect XML DTD since no XML element may have more than one attribute of type ID.

One possible solution would be to revise the mapping algorithm for fact types between a NOLOT and a LOT in the case of multiple identifying references. By introducing a dummy XML element, we can treat the involved LOT as an artificial NOLOT. Defining the value of the LOT as an attribute of type ID the attribute list of the dummy element ensures its uniqueness in a validated XML document based on the DTD. Using this technique, the following (partial) XML DTD would be created:

```
<!ELEMENT Member (has)>
<!ATTLIST Member memberID ID #REQUIRED>

<!ELEMENT dummy (is_of)?>
<!ATTLIST dummy worldranking ID #REQUIRED>

<!ELEMENT has EMPTY>
<!ATTLIST has worldranking IDREF #REQUIRED>

<!ELEMENT is_of EMPTY>
<!ATTLIST is_of memberID IDREF #REQUIRED>
```

The dummy element causes the XML DTD to be less readable because its name is artificial and thereby lacks a meaningful link to its content. Next to this, the above construction will cause a significant amount of redundant information on validated XML documents. If one wants to use the element *Member*, he must be sure that a dummy element, holding a worldranking linked to the associated memberID, is declared also.

Another, more transparent, solution is to export (a part of) the problem to an external integrity checking tool. Using this technique, a (partial) XML DTD as described in section 3.6.3.1 (part a) would be created:

```
<!ELEMENT Member (has)>
<!ATTLIST Member memberID ID #REQUIRED>

<!ELEMENT has (worldranking)>

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

The identifier constraint on the *is_of* role is exported to the integrity checking tool. The created XML DTD will remain readable since no artificial elements are created.

3.6.3.2 Binary fact types between NOLOT's

As described earlier, fact types between NOLOT's cause all roles to be declared as XML elements. Mapping identifier and mandatory constraints poses no problems at first sight. The XML cardinality operators can be used on both roles to completely express every combination of both constraints. Nevertheless, the following example will show that a mapping based on the above algorithm raises the issue of data redundancy once again.

Consider a UoD in which a Member is referenced by a unique memberID and a Course is referenced by a unique name. Every Member must follow exactly one Course. Different Members can not follow the same Course. Also, a Course can be followed by a maximum of one Member. In this UoD, memberID can be used to refer to a unique Course and a name can be used as a reference for a Member. The conceptual model of this UoD is presented as an ORM fact type in Figure 3.9.

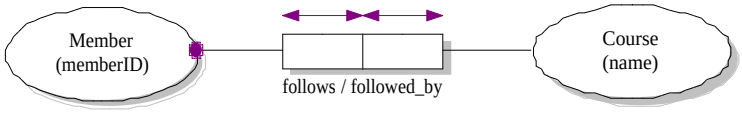


Figure 3.9: NOLOT-NOLOT fact type with a mandatory constraint and a uniqueness constraint on both roles.

Mapping this ORM fact type by use of the above algorithm would result in the following XML DTD (named teaching.dtd):

```
<?xml version=”1.0”?>
<!DOCTYPE Personal_Teaching [

<!ELEMENT Member (follows)>
<!ATTLIST Member memberID ID #REQUIRED>

<!ELEMENT Course (followed_by)?>
<!ATTLIST Course name ID #REQUIRED>

<!ELEMENT follows EMPTY>
<!ATTLIST follows name IDREF #REQUIRED>

<!ELEMENT followed_by EMPTY>
<!ATTLIST followed_by memberID IDREF #REQUIRED>

]>
```

This XML DTD completely reflects the semantics of the CS in Figure 3.9. However, depending on the intentions of the designer, the generated XML DTD can force the creator of XML documents based on our DTD to declare a significant amount of redundant data. In our example, instances of both *Member* and *Course* have to be created in order to obtain a valid XML document. The following XML page can be validated based on our XML DTD[35]**:**

```
<?xml version=”1.0” standalone=”no”?>
<!DOCTYPE Personal_Teaching SYSTEM “teaching.dtd”>

<Personal_Teaching>

<Member memberID=”100”>
  <follows name=”Introduction to Programming”/>
</Member>

<Member memberID=”101”>
  <follows name=”Advanced Programming”/>
</Member>

<Course name=”Introduction to Programming”>
  <followed_by memberID=”100”/>
</Course>

<Course name=”Advanced Programming”>
  <followed_by memberID=”101”/>
</Course>

</Personal_Teaching>
```

[35] If a validating parser is used, the “xml:lang” attribute (like all other attributes) must be declared in the DTD. This attribute is omitted in our example for brevity.

If a designer only intends to create instances of the XML element *Member*, he still needs to declare all instances of the XML element *name* that are referred to in the attribute list of the XML element *follows*. Neglecting this obligation would cause the XML document to be invalid. Since *name* is defined as an attribute of type IDREF in the attribute list of the XML element *Member*, an element bearing an ID attribute possessing the same value as the IDREF attribute must be declared in the XML document.

This raises the issue of Mayor Object Types (MOT's) in CS's. Intuitively, MOT's are the most important Object Types in a conceptual model. The identification of MOT's in a CS can be automated using a rather complex algorithm[36]. This algorithm is presented in Appendix 3. It is strongly suggested that the designer be given the option to adjust the anchors as required after automated identification. This allows additional human understanding of the UoD to impact on the final structure of the XML DTD. In the above example, a designer could explicitly express his intentions by declaring the NOLOT *Member* as a MOT in the CS. This would cause our *teaching.dtd* to be modified like this:

```
<?xml version="1.0"?>
<!DOCTYPE Personal_Teaching [

<!ELEMENT Member (follows)>
<!ATTLIST Member memberID ID #REQUIRED>

<!ELEMENT follows EMPTY>
<!ATTLIST follows name CDATA #REQUIRED>

]>
```

Since this construction lacks the mutual unique relationship between a memberID and a name (Course), an external integrity check has to be executed in order to ensure the uniqueness constraint on the omitted role (i.e. *followed_by*). In compliance with the designer's intentions, a valid XML document can be created without the necessity for redundant data. Our first XML document could be reduced to:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Personal_Teaching SYSTEM "teaching.dtd">

<Personal_Teaching>

<Member memberID="100">
  <follows name="Introduction to Programming"/>
</Member>

<Member memberID="101">
  <follows name="Advanced Programming"/>
</Member>

</Personal_Teaching>
```

A good alternative to the algorithm described in Appendix 2 is to identify possible fact types that could cause redundant data automatically and let the designer decide what to do with it. Depending on his intentions, the *most suitable* XML DTD can be generated.

[36] as proposed by Linda BIRD (nee CAMPBELL), Andrew GOODCHILD, Terry HALPIN, Object-Role Modelling and XML-Schema, ER200 Conference, LNCS 1920, page 313

3.7 Empty XML elements

Now that all object types and fact types (including identifier and mandatory constraints) are mapped, some defined XML elements still have an empty content model. The next rule ensures a valid definition of such XML elements:

RULE 15: *For each defined XML element still holding no content model, complete the definition of the XML element with the word 'EMPTY'.*

This rule will cause the earlier definitions of XML elements *Day*, *Month* and *Year* to be completed like this:

```
<!ELEMENT Day EMPTY>
<!ATTLIST Day daynr ID #REQUIRED>

<!ELEMENT Month EMPTY>
<!ATTLIST Month monthnr ID #REQUIRED>

<!ELEMENT Year EMPTY>
<!ATTLIST Year yearnr ID #REQUIRED>
```

3.8 Overall result

After applying all described rules (in a consecutive way) to our sample CS, one possible output of the algorithm could be:

```
<?xml version="1.0"?>
<!DOCTYPE Academic_domain [

<!ELEMENT Member ((Professor | Student), holds, has, born_on)>
<!ATTLIST Member email ID #REQUIRED>

<!ELEMENT Date (Day, Month, Year, birthdate_of*)>
<!ATTLIST Date dummy_date ID #REQUIRED>

<!ELEMENT Professor (teaches*)>
<!ATTLIST Professor professorID ID #REQUIRED>

<!ELEMENT Student (follows+)>
<!ATTLIST Student studentID ID #REQUIRED>

<!ELEMENT Course (teached_by, followed_by*)>
<!ATTLIST Course code ID #REQUIRED>

<!ELEMENT Day EMPTY>
<!ATTLIST Day daynr ID #REQUIRED>

<!ELEMENT Month EMPTY>
<!ATTLIST Month monthnr ID #REQUIRED>
```

```
<!ELEMENT Year EMPTY>
<!ATTLIST Year yearnr ID #REQUIRED>

<!ELEMENT membercode (#PCDATA)>
<!ELEMENT name (#PCDATA)>

<-- unnamed Fact Type -->
<-- NOLOT Member and NOLOT Date -->

<!ELEMENT born_on EMPTY>
<!ATTLIST born_on dummy_date IDREF #REQUIRED>
<!ELEMENT birthdate_of EMPTY>
<!ATTLIST birthdate_of email IDREF #REQUIRED>

<-- unnamed Fact Type -->
<-- NOLOT Professor and NOLOT Course -->

<!ELEMENT teaches EMPTY>
<!ATTLIST teaches code IDREF #REQUIRED>
<!ELEMENT teached_by EMPTY>
<!ATTLIST teached_by professorID IDREF #REQUIRED>

<-- unnamed Fact Type -->
<-- NOLOT Student and NOLOT Course -->

<!ELEMENT follows EMPTY>
<!ATTLIST follows code IDREF #REQUIRED>
<!ELEMENT followed_by EMPTY>
<!ATTLIST followed_by studentID IDREF #REQUIRED>

<-- unnamed Fact Type -->
<-- NOLOT Member and LOT membercode -->

<!ELEMENT holds (membercode)>

<-- unnamed Fact Type -->
<-- NOLOT Member and LOT name -->

<!ELEMENT has (name)>

]>
```

As described in section 3.6, the output of the algorithm can be different based on the intentions of the designer. In any case, the algorithm should produce the *best suitable* XML DTD.

3.9 Problems when mapping ORM CS's to XML DTD's

3.9.1 Most constraints in ORM CS's can not be mapped

Since ORM (or better: FORML) is by far a richer programming language than XML, most constraints in ORM CS's can not be mapped to XML DTD's and are therefore exported to application software. Consider the following non-exhaustive list of problem fields:

- Disjunctive mandatory constraints
- Disjunctive references
- Textual subtype constraints
- Ring constraints for binary facts
- External identifier constraints
- Range and subrange constraints
- Patterns
- Role frequency constraints
- Role subset/equality constraints
- Fact subset/equality constraints
- Role exclusion constraints
- Fact exclusion constraints
- Path implication and equivalence
- State transition constraints
- General RIDL procedural constraints
- ...

Mapping all these constraints to application software requires advanced programming skills and expertise understanding of ORM CS's and XML DTD's.

3.9.2 Naming and whitespaces

ORM allows whitespaces in all names while XML does not allow whitespaces. This is of course a minor problem that can easily be solved by replacing all whitespaces in ORM names by the underscore character.

3.9.3 Restrictions on rolenames

As described in section 3.7, most roles appearing in CS's will be mapped to XML elements. Since all XML elements defined in an XML DTD must have distinct names, all rolenames must have distinct names also. This restriction does not hold for roles appearing in non-explicit references because these roles are not mapped to XML elements.

3.9.4 XML related problems

As described in section 2.7.2, XML 1.0 suffers from various important shortcomings, such as syntactic complexity and lack of support for datatyping, namespacing and inheritance. These shortcomings have posed serious constraints during the elaboration process of our algorithm. Since the W3C XML Schema Working Group has addressed most of these constraints, the next logical step in this context would be to elaborate an algorithm that maps ORM CS's to XML Schema constructs. An effort in this field has been done by Linda BIRD, Andrew GOODCHILD and Dr. Terry HALPIN[37]. Of course, a different approach would be to use another conceptual modeling technique. Rainer CONRAD, Dieter SCHEFFNER and J. Christoph FREYTAG have tried to use UML as the basic conceptual modeling technique[38].

[37] Linda BIRD (nee CAMPBELL), Andrew GOODCHILD, Terry HALPIN, *Object-Role Modelling and XML-Schema*, 19th International Conference on Conceptual Modeling (ER200 Conference), Salt Lake City, Utah, USA, October 2000, Proceedings, Lecture Notes in Computer Science, Vol. 1920, Springer

[38] Rainer CONRAD, Dieter SCHEFFNER, J. Christoph FREYTAG, *XML Conceptual Modeling Using UML*, 19th International Conference on Conceptual Modeling (ER200 Conference), Salt Lake City, Utah, USA, October 2000, Proceedings, Lecture Notes in Computer Science, Vol. 1920, Springer

3.10 Quick summary

For the sake of clarity, we hereby present a quick summary of all steps in the elaborated algorithm:

RULE 1: *Determine a primary reference for each non-subtype NOLOT. Mark all involved object types as "used". For non-explicit references, also mark all involved roles as "used".*

RULE 2: *Define each non-subtype NOLOT with a unary primary reference as an XML element. If the primary reference is lexical, use it to define a #REQUIRED attribute of type ID for this XML element. Else, use it to define a #REQUIRED attribute of type IDREF for this XML element.*

RULE 3: *Define each non-subtype NOLOT with a non-unary primary reference as an XML element with all parts of the primary reference in its content model. Use a dummy name to define a #REQUIRED attribute of type ID for this XML element.*

RULE 4: *If possible, determine a primary reference for each subtype NOLOT. Mark all involved object types as "used". For non-explicit references, also mark all involved roles as "used". If no primary reference is found, use the primary reference of the supertype NOLOT as the primary reference for the subtype NOLOT.*

RULE 5: *Define each subtype NOLOT with a unary primary reference as an XML element. If the primary reference is lexical, use it to define a #REQUIRED attribute of type ID for this XML element. Else, use it to define a #REQUIRED attribute of type IDREF for this XML element.*

RULE 6: *Define each subtype NOLOT with a non-unary primary reference as an XML element with all parts of the primary reference in its content model. Use a dummy name to define a #REQUIRED attribute of type ID for this XML element.*

RULE 7: *For each supertype NOLOT, build a list of its subtypes. Depending on the exclusion and totality constraints, structure this list as follows:*

> *a. Total and Exclusive:*
> *Use the '|' separator between all subtypes in the list.*
>
> *b. Not Total and Exclusive:*
> *Add the '?' cardinality operator after all subtypes in the list and use the '|' separator between all subtypes in the list.*
>
> *c. Total and non-Exclusive:*
> *If a supertype has n different subtypes, build all possible distinct lists of n different subtypes with (n-1) subtypes holding the '?' cardinality operator. Make a '|'-separated list of all constructed lists.*
>
> *d. Not Total and non-Exclusive:*
> *Add the '?' cardinality operator after all subtypes in the list and use the ',' separator between two or more subtypes in the list.*

Add the constructed list to the content model of the involved supertypes.

RULE 8: *Define each non-marked LOT as an XML element of type #PCDATA. Mark all involved LOT's as being "used".*

RULE 9: *For unnested fact types between NOLOT's, define all non-marked roles as EMPTY XML elements. Use the (ID reference of the) co-NOLOT to add a #REQUIRED attribute of type IDREF to this XML element.*

RULE 10: *For unnested fact types between a NOLOT and a LOT, define the non-marked role played by the NOLOT as an XML element with the name of the involved LOT in its content model.*

RULE 11 (together with Rule 9 and 10): *If a role in an unnested fact type is played by a NOLOT, add the rolename to the content model of the XML element defined earlier based on this NOLOT.*

RULE 12: *For nested fact types between NOLOT's, use the name of the nested fact type to define an XML element holding both involved rolenames in its content model. Use a dummy name to define a #REQUIRED attribute of type ID for this XML element. Consider this new XML element as (mapped from) an artificial NOLOT and use the Rules for unnested fact types to map all fact types this artificial NOLOT is involved in.*

RULE 13: *For nested fact types between a NOLOT and a LOT, use the name of the nested fact type to define an XML element holding the name of the role played by the NOLOT in its content model. Use a dummy name to define a #REQUIRED attribute of type ID for this XML element. Use (a reference to) the NOLOT to define a #REQUIRED attribute of type IDREF for this XML element. Consider this new XML element as (mapped from) an artificial NOLOT and use the Rules for unnested fact types to map all fact types this artificial NOLOT is involved in.*

RULE 14: *Add identifier and mandatory constraints using the XML cardinality operators. Mark all roles and nested fact types as being "used".*

RULE 15: *For each defined XML element still holding no content model, complete the definition of the XML element with the word 'EMPTY'.*

4. TransORM: Implementing the elaborated algorithm

4.1 Introduction

This section describes the TransORM program. This program implements the algorithm elaborated in section 3. It has been mainly programmed in ANSI C++, using the Borland C++ Builder (version 4) as the programming environment and the STL[39] as the library for container declarations and manipulations. It is targeted at the Windows platform (98, 2000, ME). Apart from the constructed interfaces, which hold Borland-dependant codes, all software parts are easily portable to other platforms. The program has not been built as a fully functional CASE Tool but merely as a means of demonstration for our algorithm. The source codes for the implemented mapping algorithm can be found in Appendix 4.

4.2 The TransORM database

As described in previous sections, mapping an ORM CS to an XML DTD will ignore most constraints. Therefore, we have opted to incorporate only basic features of ORM in our implementation. This choice obviously simplifies the structure of the database used to store ORM-related data. One possible database structure is presented in the next tables:

Name (NOLOT)	Reference	Parents
(primary key)		

Figure 4.1: Paradox Table for NOLOT's (Table1)

Name (LOT)	Data Type
(primay key)	

Figure 4.2: Paradox Table for LOT's (Table2)

OT1	OT2	Rolename1	Rolename2	Factname	Uniqueness	Mandatory
(primary key)						

Figure 4.3: Paradox Table for Fact Types (Table3)

The above tables, representing the TransORM database, clearly show that our database is not relational, i.e. it does not hold foreign keys. Making the database relational would of course be a possible design choice. However, it would limit our possibilities while building a CS. For example, we could define a *Reference* entry in the table for NOLOT's to be a foreign key, indicating that only (earlier declared) LOT's can be the primary reference of a NOLOT. In that case, our database would be relational, but it would not accept non-unary primary references or even NOLOT's referencing other NOLOT's. We can of course solve this problem by the construction of a far more complicated (relational) database combined with the use of an entry stack to ensure database consistency while building CS's. Since our sample implementation only intends to provide a simple demonstration of the working of our algorithm, we have chosen for a simple, non-relational database. In addition, this way of working facilitates easy understanding of the source codes.

[39] The Standard Template Library, or *STL*, is a generic C++ library of container classes, algorithms, and iterators. More info can be found at http://www.sgi.com/tech/stl/index.html

4.3 The TransORM file format

Our sample implementation supports saving and retrieving CS's to and from files. Since the Microsoft Corporation was not willing to provide us with the Visiomodeler file format for CS's, we had to invent our own file format. We have opted for the following BibTeX[40]-based format:

@NOLOT{name = … , reference = […], parents = […]}	
name	name of non-lexical object type
reference	[(list of object types constituting) the primary reference]
parents	[(list of) parent object(s)] = SUB/SUPERTYPING [] if no supertypes for Nolot
@LOT{name = … , category = …}	
name	name of lexical object type
category	label *(string)* | unit *(number)* | sign *(picture)*
@FACT{objects = […, …], rolenames = […, …], factname = …, uniqueness = …, mandatory = …}	
objects	[ObjectType1, ObjectType2] = list of two objects types playing role -> *schema MUST BE binary*
rolenames	[Rolename1, Rolename2] = list of two rolenames in fact type *same order as list of objects*
factname	name of objectified fact type [] if fact type is not objectified
uniqueness	otm *(One-to-Many)* | mto *(Many-to-One)* | oto *(One-to-One)* | mtm *(Many-to-Many)*
mandatory	mandatory constraints on all roles. YY (Yes-Yes) | YN (Yes-No) | NY (No-Yes) | NN (No-No)

Table 4.4: the TransORM file format

Appendix 5 provides a few examples of how CS's are saved to file with the above file format. Since this file format is purely character based, we can enter a CS in our program by typing a valid ORM file, saving it to disk and opening it in our program. Open/close operations on ORM files are supported by the use of guiding interfaces. Figure 4.4 shows the TransORM interface for opening an ORM file:

[40] BibTeX is a program and file format designed by Oren Patashnik and Leslie Lamport in 1985 for the LaTeX document preparation system. A complete description of the format can be found at http://www2.ecst.csuchico.edu/~jacobsd/bib/formats/bibtex.html

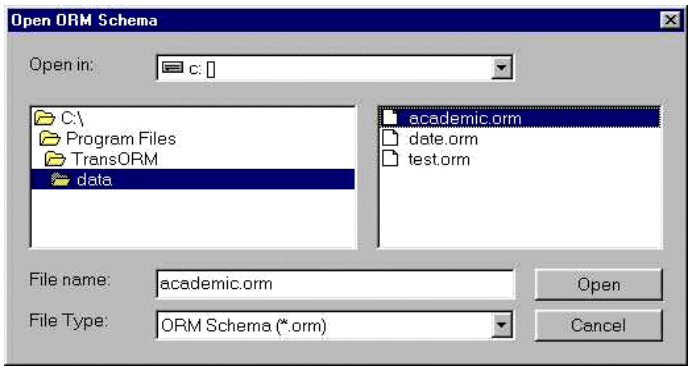


Figure 4.5: TransORM interface for opening an ORM file

4.4 The general TransORM user interface

The file format described in section 4.3 is easy-to-use for most simple CS's. However, it gets particularly cumbersome and errorprone as the complexity of a CS increases. Small mistakes like spelling errors, open braces/brackets and misplaced comma's are likely to slip in the TransORM file during a process of unguided character typing. In order to avoid this kind of errors as much as possible, our general program interface provides guided input processes on the underlying database tables. A fullscreen snapshot of the general TransORM interface is provided in figure 4.5:

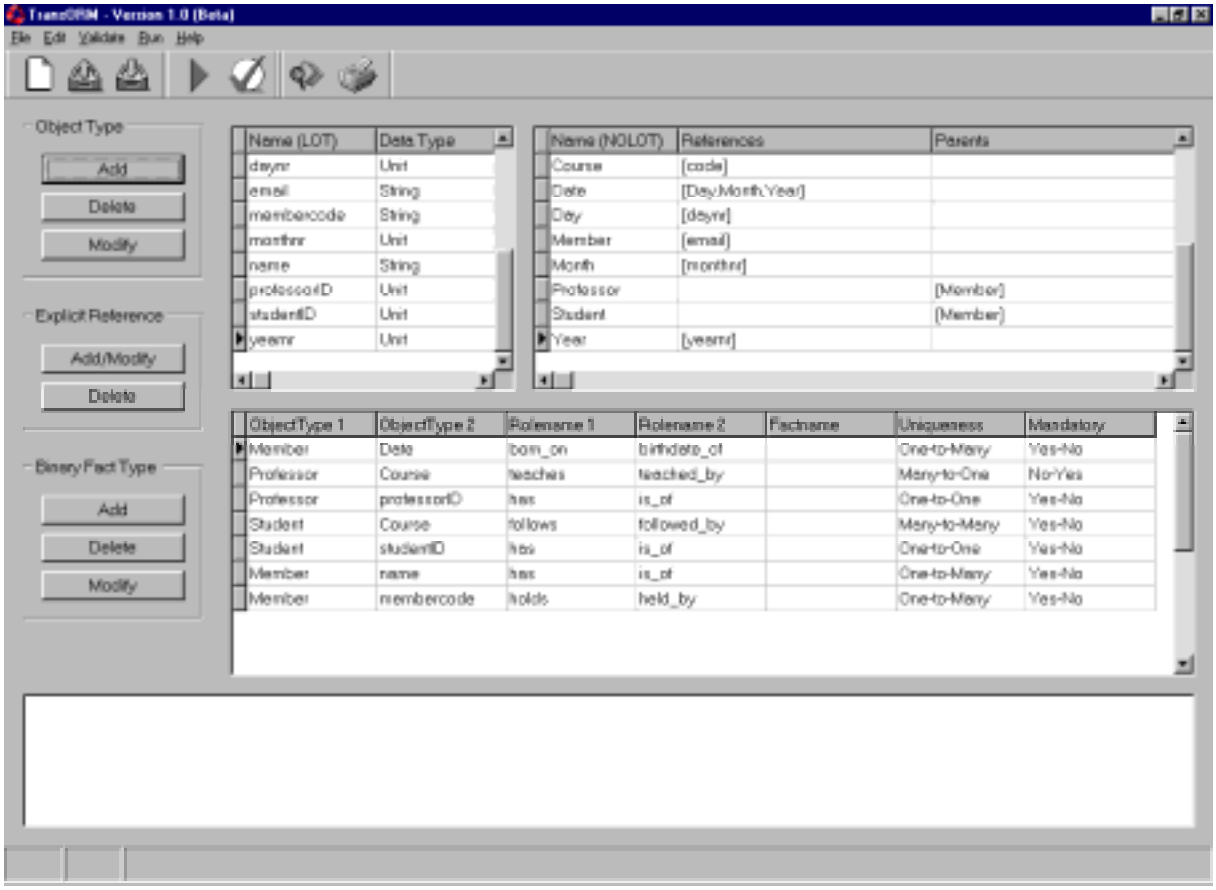


Figure 4.5: the general TransORM user interface

Although our general interface offers an open view of the content of all tables, users can not modify the content of any table in a direct way. Instead, the program interface incorporates several input interfaces, which guide database manipulations such as *add*, *delete* and *modify*. One such manipulation interface is shown in figure 4.6:

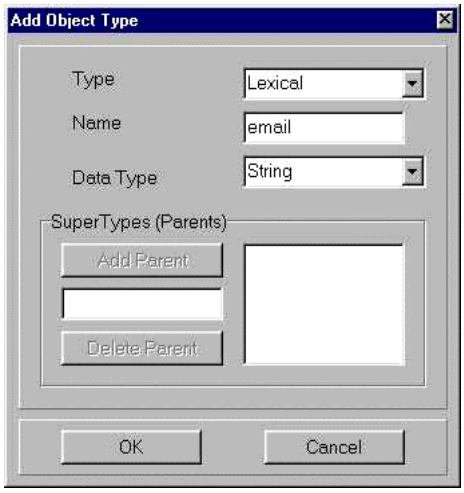


Figure 4.6: sample TransORM database manipulation interface

Guiding database manipulations does not ensure the validity of constructed CS's. Therefore, all CS's should be validated using a built-in validating tool. Since we have assumed all CS's to be valid (cfr. section 3.1), we have only provided a gateway to this validating tool. Hence, invalid database entries will cause the outcome of the TransORM mapping algorithm to be unreliable.

4.5 The TransORM converting module

4.5.1 General name and root element of the XML DTD

Once the complete CS is constructed, the TransORM user can run the converting module in order to construct a DTD. This module uses the content of all tables as the input for the implemented mapping algorithm. Before the actual execution of the algorithm starts, the converter will ask to provide the name of the DTD and the name of the root element. This is done by the following interface:

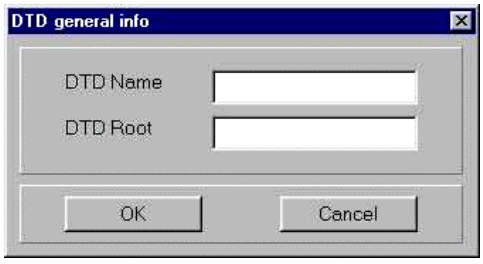


Figure 4.7: the TransORM interface for DTD general info

Although TransORM provides a separate gateway to incorporate a validating tool in the future, it would be sensible to auto-call this validating tool whenever the converting module is called. By allowing only valid CS's to be mapped, TransORM would be able to ensure absolute validity of its constructed DTD's.

4.5.2 Implementing the mapping algorithm

4.5.2.1 Finding a primary reference for all NOLOT's

The TransORM *RefReader* object (cfr. addendum for source codes) implements the following primary reference metascheme for NOLOT's by the use of the *find_primary_refs()* public member function:

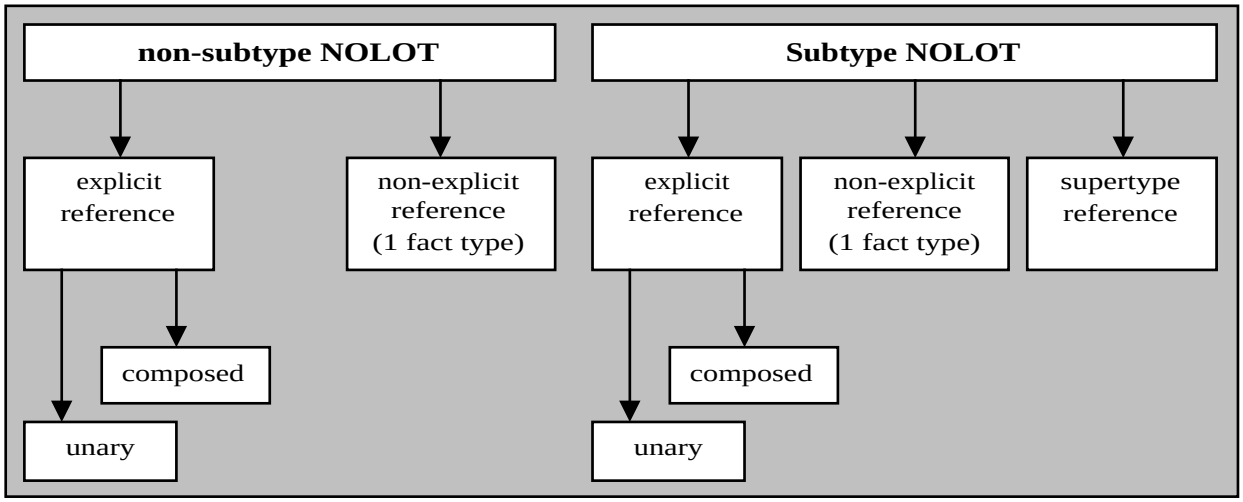


Figure 4.8: the TransORM primary reference metascheme

The above figure shows that non-subtype NOLOT's must have either an explicit or a non-explicit primary reference. In TransORM, an explicit primary reference always gets priority over a non-explicit one. If no explicit reference was found for a NOLOT, the program will search for a fact type expressing a one-to-one relationship between the involved NOLOT and another object type in the table for fact types (cfr. figure 5.3). If such a fact type is found, the other object type in the fact type will be considered to be a non-explicit primary reference for the involved NOLOT.

An explicit primary reference is either unary or composed. There is a small but important difference between composed primary references in ORM and TransORM CS's. In our sample ORM CS (cfr. section 1.4.2), NOLOT *Date* is referenced by a unique combination of a *Day*, a *Month* and a *Year*. This is indicated by placing an external identifier constraint on the roles played by NOLOT's *Day*, *Month* and *Year*. In ORM CS's, composed primary references are always non-explicit. In other words, the VisioModeler validating and mapping tool must calculate itself a combination of object types to be the primary reference for a NOLOT. Since TransORM does not support external identifier constraints, composed primary references always have to be explicit in a TransORM CS.

Once TransORM has determined a primary reference for all non-subtype NOLOT's, it will try to do the same for subtype NOLOT's. If the program does not find a primary reference for a subtype NOLOT, it will take the primary reference of the supertype NOLOT to be also the primary reference of the subtype NOLOT.

It is important to mention that the *RefReader* object manages four local STL containers: a vector that holds unused LOT's, a vector that holds unused fact types, a map that links NOLOT's and their primary reference and a map that links super- and subtypes. The first two containers are initialized and filled by the constructor of the *RefReader* object. Calling the *find_primary_refs()* public member function causes the last two containers to be filled. While executing this function, all involved LOT's and fact types are removed from their respective container in memory.

4.5.2.2 Building partial DTD fragments for NOLOT's, LOT's and Fact Types

The very core of the implementation of our mapping algorithm in TransORM is the *DtdEncoder* object (cfr. addendum for source codes). It uses a local (reference to a) *RefReader* object in its constructor in order to be able to find a primary reference for all NOLOT's and to perform additional checks (e.g. type checks, checking object types in nested fact types, etc.) on object types in the TransORM CS. Basically, the constructor of the *DtdEncoder* object builds a local STL map in memory, holding the name of all XML elements, their attribute list and their content model. It does so in a pre-defined order: first NOLOT's, then LOT's and finally Fact Types.

For NOLOT's, it starts by finding all primary references. This is done by calling the *find_primary_refs()* public member function on the local *RefReader* object, referring to Rule 1 and 4 of our mapping algorithm. Next, it uses the content of the *RefReader* local STL map containing primary references, combined with some of the *RefReader* public member (checking) functions, to insert XML element definitions, attribute list definitions and content models for all NOLOT's in the local STL map. This step refers to Rule 2, 3, 5 and 6 of our algorithm. Referring to Rule 7 of our algorithm, the mapping of NOLOT's is finalized by completing the content model of supertype NOLOT's, using the *RefReader* local STL map containing super- and subtype links. This version of TransORM assumes totality and exclusion constraints on subtypes for reasons of simplification.

Next, the *DtdEncoder* constructor inserts all unmarked LOT's as XML elements of type *#PCDATA* in its local STL map. In order to do so, it uses the *RefReader* local STL container holding unused LOT's. This step refers to Rule 8 of our algorithm.

Finally, the *RefReader* local STL container holding unused fact types is used by the *DtdEncoder* constructor to map all unmarked fact types, as prescribed by Rules 9 to 14.

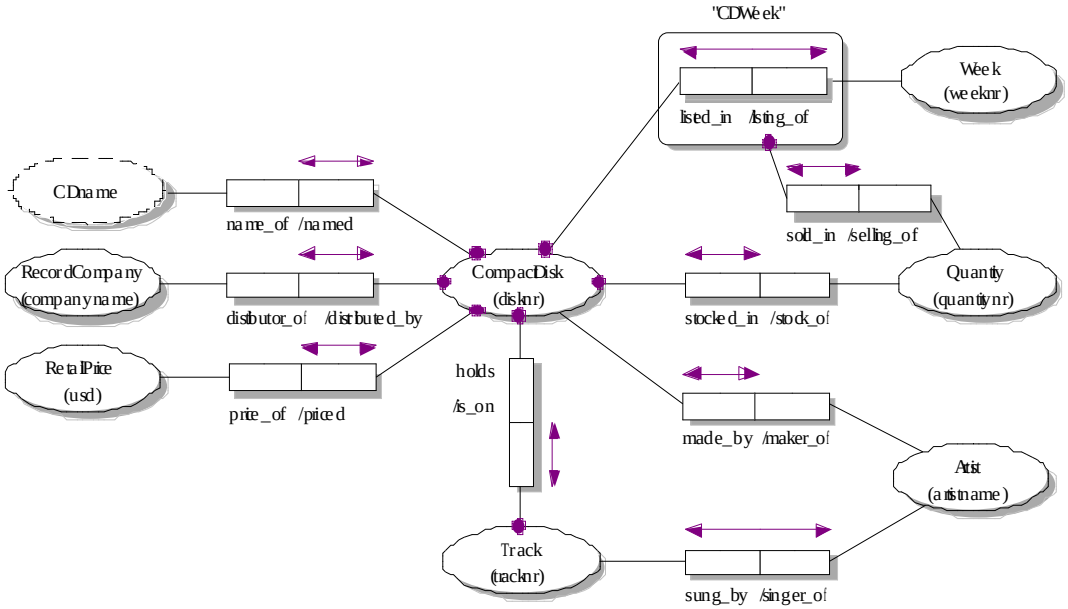
Once the *DtdEncoder* object is constructed, TransORM can start building the actual DTD. This is done by the *DtdBuilder* object.

4.5.2.3 Building the actual DTD

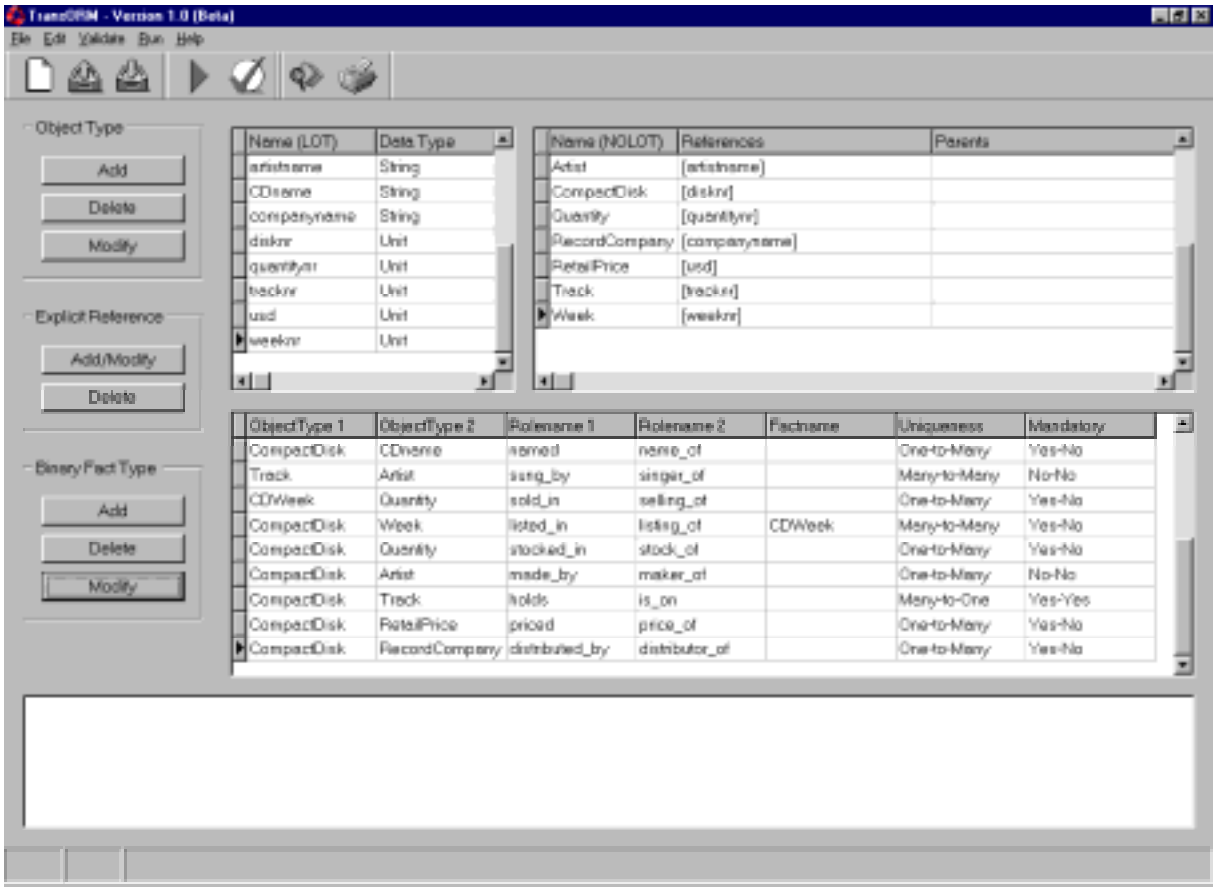
The *DtdBuilder* object constructs a local C++ *String* object, using the content of the local STL container held by the *DtdEncoder* object. Each XML element, associated attribute list and content model found in this container will be mapped to a textual representation based upon the XML syntax for DTD's and added to the *String* object. In this version of TransORM, the textual representations are added in a pre-defined order: First elements mapped from supertype NOLOT's, then those mapped from all other NOLOT's, next those based on LOT's and finally those mapped from fact types. Following Rule 15 of our algorithm, elements holding no content model will be added as *EMPTY* XML elements. Once the *String* object contains all XML elements, it is written to file. The name of the file and the root element of the constructed DTD must be provided by the TransORM user (cfr. section 4.5.2).

4.6 A TransORM example: CompactDisks

Consider the following ORM CS:



In TransORM, this ORM CS can be represented like this:



Running the TransORM converting module will result in the construction of the following DTD (input for *root element*: CompactDisks):

```
<?xml version="1.0"?>
<!DOCTYPE CompactDisks [

<!ELEMENT Artist (maker_of*,singer_of*)>
<!ATTLIST Artist artistname ID #REQUIRED>

<!ELEMENT CompactDisk
(distributed_by,priced,holds+,made_by?,stocked_in,listed_in+,named)>
<!ATTLIST CompactDisk disknr ID #REQUIRED>

<!ELEMENT Quantity (stock_of*,selling_of*)>
<!ATTLIST Quantity quantitynr ID #REQUIRED>

<!ELEMENT RecordCompany (distributor_of*)>
<!ATTLIST RecordCompany companyname ID #REQUIRED>

<!ELEMENT RetailPrice (price_of*)>
<!ATTLIST RetailPrice usd ID #REQUIRED>

<!ELEMENT Track (is_on,sung_by*)>
<!ATTLIST Track tracknr ID #REQUIRED>

<!ELEMENT Week (listing_of*)>
<!ATTLIST Week weeknr ID #REQUIRED>

<!ELEMENT CDname (#PCDATA)>

<!ELEMENT CDWeek (CompactDisk,Week,sold_in)>
<!ATTLIST CDWeek Ref ID #REQUIRED>

<!ELEMENT distributed_by EMPTY>
<!ATTLIST distributed_by companyname IDREF #REQUIRED>

<!ELEMENT distributor_of EMPTY>
<!ATTLIST distributor_of disknr IDREF #REQUIRED>

<!ELEMENT holds EMPTY>
<!ATTLIST holds tracknr IDREF #REQUIRED>

<!ELEMENT is_on EMPTY>
<!ATTLIST is_on disknr IDREF #REQUIRED>

<!ELEMENT listed_in EMPTY>
<!ATTLIST listed_in weeknr IDREF #REQUIRED>

<!ELEMENT listing_of EMPTY>
<!ATTLIST listing_of disknr IDREF #REQUIRED>

<!ELEMENT made_by EMPTY>
<!ATTLIST made_by artistname IDREF #REQUIRED>

<!ELEMENT maker_of EMPTY>
<!ATTLIST maker_of disknr IDREF #REQUIRED>

<!ELEMENT named (CDname)>
```

```
<!ELEMENT price_of EMPTY>
<!ATTLIST price_of disknr IDREF #REQUIRED>

<!ELEMENT priced EMPTY>
<!ATTLIST priced usd IDREF #REQUIRED>

<!ELEMENT selling_of EMPTY>
<!ATTLIST selling_of CDWeek_ID IDREF #REQUIRED>

<!ELEMENT singer_of EMPTY>
<!ATTLIST singer_of tracknr IDREF #REQUIRED>

<!ELEMENT sold_in EMPTY>
<!ATTLIST sold_in quantitynr IDREF #REQUIRED>

<!ELEMENT stock_of EMPTY>
<!ATTLIST stock_of disknr IDREF #REQUIRED>

<!ELEMENT stocked_in EMPTY>
<!ATTLIST stocked_in quantitynr IDREF #REQUIRED>

<!ELEMENT sung_by EMPTY>
<!ATTLIST sung_by artistname IDREF #REQUIRED>

]>
```

5. Conclusions

Throughout this thesis, we have mainly aimed our efforts at the link between ORM and XML. We have created this link by the construction of a mapping algorithm between ORM CS's and XML DTD's. Although our algorithm generates valuable output for all possible CS's, it suffers from a few important drawbacks:

- It ignores valuable information found in CS's due to the fact that the ORM syntax is much richer than the XML syntax. In this regard, we refer to the lack of support for data typing, inheritance models and most constraints found in CS's.
- Complex CS's holding a lot of fact types will cause our algorithm to generate a significant number of XML element definitions holding at least one dummy attribute.
- Without the identification of Mayor Object Types (MOT), the generated DTD's will force valid XML documents to hold a significant amount of redundant data. The most important problem here is that the identification process of MOT's is a complex matter, even for rather simple CS's.

The above problems, encountered during the mapping process, clearly obstruct proper and easy human understanding of the generated DTD's. In most cases, developers will need a full understanding of the underlying CS's in order to understand the generated DTD's. But even then, the DTD's will never reflect all semantic properties found in CS's. A lot of information and constraints will have to be exported to external integrity checking tools. At this point, one can easily question the efficiency and utility of our mapping algorithm.

It is our conviction though that our efforts have not been fruitless. In the first place, they have helped to demonstrate most important problems fields in the XML syntax. Next to this, however poor the output of our algorithm might look, it will always be possible to use it for further processing and interpretation by other tools or applications. In addition, our exercise helps to promote the idea of consistency between database structure and data structure for websites. Ensuring this consistency can contribute to improve cost- and time-efficiency in the development process of web-based software systems linked to one or more databases. We are convinced that this will be an important issue for future software engineering processes and techniques.

Considering the impressive efforts made by a wide variety of people in the fine-tuning and the extension of the XML syntax, future work in the field of linking Conceptual Modeling techniques and XML will no doubt result in better algorithms and/or tools. At the time of writing this text, academic researchers have already started to communicate their findings while examining the possible linking of ORM and XML-Schema[41].

[41] E.g. Linda BIRD (nee CAMPBELL), Andrew GOODCHILD, Terry HALPIN, *Object-Role Modeling and XML-Schema*, 19th International Conference on Conceptual Modeling (ER200 Conference), Salt Lake City, Utah, USA, October 2000, Proceedings, Lecture Notes in Computer Science, Vol. 1920, Springer

Table of References

- **ORM related hyperlinks**

Object Role Modeling
(The official site for Conceptual Data Modeling)
http://www.orm.net/

The Journal of Conceptual Modeling
http://www.inconcept.com/JCM/

- **XML related hyperlinks**

The World Wide Web Consortium (W3C) homepage
http://www.w3.org/

Extensible Markup Language 1.0 (Second Edition) - *XML*
(W3C Recommendation 6 October 2000)
http://www.w3.org/TR/2000/REC-xml-20001006

Extensible Markup Language – *detailed information*
http://www.w3.org/XML/#9802xml10

Extensible Markup Language - Activity Statement
http://www.w3.org/XML/Activity.html

XML Schema Part 0: Primer
(W3C Proposed Recommendation 30 March 2001)
http://www.w3.org/TR/xmlschema-0/

XML Schema Part 1: Structures
(W3C Proposed Recommendation 30 March 2001)
http://www.w3.org/TR/xmlschema-1/

XML Schema Part 2: Datatypes
(W3C Proposed Recommendation 30 March 2001)
http://www.w3.org/TR/xmlschema-2/

XML-Data
(W3C Note 5 January 1998)
http://www.w3.org/TR/1998/NOTE-XML-data-0105/Overview.html

Schema for Object-Oriented XML 2.0 - *SOX*
(W3C Note 30 July 1999)
http://www.w3.org/TR/NOTE-SOX/

Document Content Description for XML – *DCD*
(W3C Submission 31 July 1998)
http://www.w3.org/TR/NOTE-dcd

Document Definition Markup Language Specification, Version 1.0 - *DDML*
(W3C Note, 19 January 1999)
http://www.w3.org/TR/NOTE-ddml

Cascading Style Sheets, level 1 – *CSS1*
(W3C Recommendation 17 Dec 1996, revised 11 Jan 1999)
http://www.w3.org/TR/REC-CSS1

Cascading Style Sheets, level 2 – *CSS2*
(W3C Recommendation 12 May 1998)
http://www.w3.org/TR/REC-CSS2/

Extensible Stylesheet Language (Version 1.0) - *XSL*
(W3C Candidate Recommendation 21 November 2000)
http://www.w3.org/TR/xsl/

Namespaces in XML
(W3C 14 January 1999)
http://www.w3.org/TR/REC-xml-names/

Document Object Model (Technical Reports) - *DOM*
http://www.w3.org/DOM/DOMTR

Document Content Description for XML - *DCD*
(Submission to the W3C 31 July 1998)
http://www.w3.org/TR/NOTE-dcd

Schema for Object-Oriented XML 2.0 - *SOX*
(W3C Note 30 July 1999)
http://www.w3.org/TR/NOTE-SOX/

Document Definition Markup Language (Specification, Version 1.0) - *DDML*
(W3C Note 19 January 1999)
http://www.w3.org/TR/NOTE-ddml

XML Linking Language (Version 1.0) - *XLink*
(W3C Proposed Recommendation 20 December 2000)
http://www.w3.org/TR/xlink/

XML Base
(W3C Proposed Recommendation 20 December 2000)
http://www.w3.org/TR/xmlbase/

XML Pointer Language (Version 1.0) - *XPointer*
(W3C Last Call Working Draft 8 January 2001)
http://www.w3.org/TR/WD-xptr

XML Path Language (Version 1.0) - *XPath*
(W3C Recommendation 16 November 1999)
http://www.w3.org/TR/xpath

A Query Language for XML - *XQuery*
(W3C Working Draft 15 February 2001)
http://www.w3.org/TR/xquery/

XML Signature Working Group
http://www.w3.org/Signature/

XML Encryption Working Group
http://www.w3.org/Encryption/2001/

- **Other hyperlinks**

Standard Template Library Programmer's Guide
http://www.sgi.com/tech/stl/index.html

BibTex program and file format
http://www2.ecst.csuchico.edu/~jacobsd/bib/formats/bibtex.html

- **Books**

Bjarne STROUSTROUP, 1998, The C++ Programming Language (3rd Edition), Addison-Wesley (ISBN 0-201-88954-4)

Didier MARTIN, Mark BIRBECK, Michael KAY, e.a., 2000, Professional XML, Wrox Press, USA (ISBN 1-861003-11-0)

James COHOON, Jack DAVIDSON, 1999, C++ programming Design: an introduction to programming and Object-Oriented Design (2nd Edition), The McGraw-Hill Companies (ISBN 0-07-116147-3)

Kent REISDORPH, 1999, Borland C++ Builder 4 Unleashed, Sams Publishing, USA (ISBN 0-672-31510-6)

Martin FOWLER, Kendall SCOTT, 1997, UML distilled: applying the standard object modeling language, Addison-Wesley (ISBN 90-430-0199-6)

Terry HALPIN, 1999, Conceptual Schema and Relational Database Design (2nd Edition), WytLytPub, Bellevue, USA

Terry HALPIN, 2001, Information Modeling and Relational Databases, Morgan Kaufmann Publishers (ISBN 1-55860-672-6).

Thomas CONNOLLY, Carolyn BEGG, Anne STRACHAN, 1998, Database Systems (2nd Edition), Addison-Wesley (ISBN 0-201-34287-1)

Simon NORTH, Paul HERMANS, 1999, Sams Teach Yourself XML in 21 days, Sams Publishing, USA (ISBN 1-57521-396-6)

- **Articles and papers**

Gary F. SIMONS, 1994, *Conceptual modeling versus visual modeling: a technological key to building consensus*, Dallas, Summer Institute of Linguistics – A paper presented at: Consensus ex Machina, Joint International Conference of the Association for Literary and Linguistic Computing and the Association for Computing and the Humanities, Paris, 19-23 April 1994 - paper presented at http://www.sil.org/cellar/ach94/ach94.html

G. P. BAKEMA, J. P. C. ZWART, H. VAN DER LEK, 1993, *Fully Communication Oriented NIAM* - paper presented at http://www.fcoim.com/FCONIAM.HTM

J. C. NORDBOTTEN, Martha E. CROSBY, *Recognising Graphic Detail - An Experiment in User Interpretation of Data Models*, 13th British National Conference on Databases, BNCOD 13, Manchester, United Kingdom, July 12-14, 1995, Proceedings, Lecture Notes in Computer Science, Vol. 940, Springer, ISBN 3-540-60100-7

Linda BIRD (nee CAMPBELL), Andrew GOODCHILD, Terry HALPIN, *Object-Role Modeling and XML-Schema*, 19th International Conference on Conceptual Modeling (ER200 Conference), Salt Lake City, Utah, USA, October 2000, Proceedings, Lecture Notes in Computer Science, Vol. 1920, Springer

Niklaus WIRTH, *What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions*, Communications of the ACM, Vol. 20, number 11, november 1997

Peter CHEN, March 1976, *The Entity-Relationship Model -- Toward a Unified View of Data,* ACM Transactions on Database Systems, Vol. 1, No. 1, pages 9 to 36 – paper presented at http://www.csc.lsu.edu/~chen/chen.html

Rainer CONRAD, Dieter SCHEFFNER, J. Christoph FREYTAG, *XML Conceptual Modeling Using UML*, 19th International Conference on Conceptual Modeling (ER200 Conference), Salt Lake City, Utah, USA, October 2000, Proceedings, Lecture Notes in Computer Science, Vol. 1920, Springer

Appendices

Appendix 1:

A formal mapping algorithm to create a physical database schema from an ORM diagram. (by Prof. Dr. Robert Meersman)

0. Assume ORM diagram to be binary

1. Check referential completeness

2. Grouping around non-subtype NOLOT's

   a. for each NOLOT A, take all <A r_i r_i' B> fact types where r_i is identifying and unused. Construct a pre-table by putting them all in a sequence.

      A: B_1 r_1' B_2 r_2' B_3 r_3' …

   b. if r_i' is identifying, mark it with a closed arrow.

      A: B_1 r_1' B_2 r_2' B_3 r_3' … $\overleftrightarrow{B_i\ r_i'}$ ...

   c. if there exists a uniqueness constraint on r_{i1}' to r_{ik}', mark it with open arrows.

      A: … $\overleftarrow{B_{i1}\ r_{i1}'}$ … $\overrightarrow{B_{ik}\ r_{ik}'}$ …

   d. if ri' is not mandatory, mark it by putting it between parentheses.

      A: B_1 r_1' B_2 r_2' B_3 r_3' … (B_i r_i') …

   e. if B_i is a NOLOT, mark it by underlining it.

      A: B_1 r_1' B_2 r_2' B_3 r_3' … $\underline{B_i\ r_i'}$ ...

   f. mark all r_i' appearing in the constructed sequences as 'used' in the ORM diagram

3. Grouping around subtype NOLOT's

   Perform steps 2a. to 2f. on each subtype NOLOT C, where C ≠ A.
   Call those steps 3a. to 3f.

   g. add any attribute A, where C ≠ A, to the pre-table as identifying.

      C: $\overleftrightarrow{A}$ B_1 r_1' B_2 r_2' B_3 r_3' …

4. Group unused fact types in their own (binary) pre-table.

5. Make all constructed pre-tables lexical. For each $\underline{B_i\ r_i'}$, substitute NOLOT B by a lexical reference to it.

6. Add remaining constraints (if possible). Carefully walk through the ORM diagram and check if every constraint has been represented.

7. Eliminate unwanted reference tables (= tables only consisting of an identifier).

Appendix 2:

XML Schema built-in datatypes and their fundamental facets

	Datatype	ordered	bounded	cardinality	numeric
Primitive Datatypes	string	false	false	countably infinite	false
	boolean	false	False	finite	false
	float	true	true	finite	true
	double	true	true	finite	true
	decimal	true	false	countably infinite	true
	timeDuration	true	false	countably infinite	false
	recurringDuration	true	false	countably infinite	false
	binary	false	false	countably infinite	false
	uriReference	false	false	countably infinite	false
	ID	true	false	countably infinite	false
	IDREF	true	false	countably infinite	false
	ENTITY	true	false	countably infinite	false
	QName	true	false	countably infinite	false
Derived Datatypes	CDATA	false	false	countably infinite	false
	token	false	false	countably infinite	false
	language	false	false	countably infinite	false
	IDREFS	false	false	countably infinite	false
	ENTITIES	false	false	countably infinite	false
	NMTOKEN	false	false	countably infinite	false
	NMTOKENS	false	false	countably infinite	false
	Name	false	false	countably infinite	false
	NCName	false	false	countably infinite	false
	NOTATION	true	false	countably infinite	false
	integer	true	false	countably infinite	true
	nonPositiveInteger	true	false	countably infinite	true
	negativeInteger	true	false	countably infinite	true
	long	true	true	finite	true
	int	true	true	finite	true
	short	true	true	finite	true
	byte	true	true	finite	true
	nonNegativeInteger	true	false	countably infinite	true
	unsignedLong	true	true	finite	true
	unsignedInt	true	true	finite	true
	unsignedShort	true	true	finite	true
	unsignedByte	true	true	finite	true
	positiveInteger	true	false	countably infinite	true
	timeInstant	true	false	countably infinite	false
	time	true	false	countably infinite	false
	timePeriod	true	false	countably infinite	false
	date	true	false	countably infinite	false
	month	true	false	countably infinite	false
	year	true	false	countably infinite	false
	century	true	false	countably infinite	false
	recurringDate	true	false	countably infinite	false
	RecurringDay	true	false	countably infinite	false

Appendix 3:

An algorithm that identifies Mayor Object Types in an ORM CS.

1. Any Fact Type role involved in a non-implied mandatory role constraint is weighted in inverse proportion to the number of roles participating in the constraint.

2. The player of the role in a unary predicate is 'the most important participant' in that predicate, and is weighted accordingly.

3. If only one role in a Fact Type is played by a 'non-leaf' Object Type, then this role is 'conceptually important' enough to be given a strong weighting.

4. If exactly one role within a Fact Type has the 'smallest maximum frequency' (calculated based on both uniqueness and frequency constraints) of that Fact Type, this role should be anchored.

5. If exactly one role in a Fact Type is played by a non-value type, then the Fact Type should be anchored on this role.

6. If exactly one role in a given Fact Type is played by an Object Type that became an anchor point via rules 1 to 5, the Fact Type is anchored on this role.

7. If a Fact Type is involved in exactly one single-role set constraint (i.e. subset, equality or exclusion), and the role at the other end of the set constraint is anchored, then the constrained role in the given Fact Type should also be anchored.

8. If a Fact Type is involved in exactly one (possible multi-role) set constraint and exactly one of the roles in the Fact Type is in the corresponding position within the set constraint as an anchored role, then this role is itself anchored.

9. If there exists a non-implied set constraint in which one of the roles involved in the constraint is the only involved role in its Fact Type to be played by an anchor point and the corresponding role's Fact Type in the other role sequence is not anchored, then this becomes an anchor.

10. Those unanchored Fact Types, in which only one role is the 'join role' for some set constraint role sequence, should be anchored on this 'join role'.

11. The first role of each multi-role, non-implied set constraint becomes an anchor, if its Fact Type is not already anchored.

12. Any Fact Type not already anchored should be anchored on the first role involved in an internal uniqueness constraint.

After these 'anchoring' rules are applied, the Mayor Object Types are identified as those Object Types to which some Fact Type is anchored.

Appendix 4:

Source codes for the TransORM mapping algorithm

RefReader.h

```
#ifndef REFREADER_H
#define REFREADER_H

#include <String.h>
#include <map.h>
#include <vector.h>

typedef vector<String>                      s_vector;
typedef vector<s_vector>                    sv_vector;
typedef sv_vector::iterator                 sv_iter;
typedef map<String,s_vector>                svs_map;
typedef svs_map::iterator                   svs_iter;
typedef map<String,String>                  ss_map;
typedef ss_map::iterator                    ss_iter;
typedef pair<String,String>                 ss_pair;
class RefReader
{
public:
        RefReader();
        ~RefReader();

        typedef pair<String,String>         ss_pair;
        typedef multimap<String,ss_pair>    smap;
        typedef smap::iterator              siter;
        typedef  smap::const_iterator       iterator;

        void            find_primary_refs();
        sv_vector&      get_unmarked_lots();
        sv_vector&      get_unmarked_facts();
        svs_map&        get_inheritance();
        String          find(String nolot);
        void            replace(String nolot,String dummy_ref);
        String          find_type(String object_name);
        ss_pair         find_nested_objects(String fact_name);
        bool            empty() const;

        iterator  begin() const;
        iterator  end() const;

private:
        smap            primrefs_;
        sv_vector       facts_;
        sv_vector       lots_;
        svs_map         supertypes_;

        void            insert_ref(String& base_nolot,String& ref,bool unary);
        void            mark_lot_as_used(String& lot);
        void            insert_subtype(String& supertype,String subtype);
};

#endif
```

RefReader.cpp

```
#include <vcl.h>
#include "RefReader.h"
#include "Main.h"
#include "MyString.h"
```

```
#pragma hdrstop

RefReader::RefReader()
{
TTable *l_tab = Form1->Table2; //table of Lot's
TTable *f_tab = Form1->Table3; //table of Facts
//build 'fact type' vector in memory
vector<String> fact(7);
f_tab->First();
while(!f_tab->Eof) {
        for (unsigned int i=0;i<7;++i)
                fact[i]=f_tab->Fields->Fields[i]->AsString;
        facts_.push_back(fact);
        f_tab->Next();
        }
//build 'lexical object type' vector in memory
vector<String> lot(2);
l_tab->First();
while(!l_tab->Eof) {
        for (unsigned int i=0;i<2;++i)
                lot[i]=l_tab->Fields->Fields[i]->AsString;
        lots_.push_back(lot);
        l_tab->Next();
        }
}

void
RefReader::find_primary_refs()
{
TTable *n_tab = Form1->Table1; //table of Nolot's
ss_map unref_nolots;
//check references of non-subtype Nolot's
n_tab->First();
while(!n_tab->Eof){
        String base(n_tab->Fields->Fields[0]->AsString); //Nolot name
        MyString target(n_tab->Fields->Fields[1]->AsString); //explicit reference
        String raw_target(target.unwrap()); //remove brackets;
        //mark supertypes
        MyString parent(n_tab->Fields->Fields[2]->AsString);
        String raw_parent(parent.unwrap()); //remove brackets
        if (!raw_parent.IsEmpty()) this->insert_subtype(raw_parent,base);
        //explicit unary reference
        if (!raw_target.IsEmpty()) {
                if (!target.is_composed())
                        this->insert_ref(base,raw_target,true);
                else
                        this->insert_ref(base,raw_target,false);
                }
        //non-explicit unary reference
        else if (raw_target.IsEmpty()) {
                sv_iter it=facts_.begin();
                for (it;it!=facts_.end();++it) {
                        String OT1((*it)[0]),OT2((*it)[1]),ident((*it)[5]),mand((*it)[6]),non_exp_ref;
                        if ((OT1==base)&&(ident=="One-to-One")&&
                           (mand=="Yes-No"||mand=="Yes-Yes")) non_exp_ref=OT2;
                        else if ((OT2==base)&&(ident=="One-to-One")&&
                              (mand=="No-Yes"||mand=="Yes-Yes")) non_exp_ref=OT1;
                        if (!non_exp_ref.IsEmpty()) {
                                facts_.erase(it);//mark all roles as used
                                this->insert_ref(base,non_exp_ref,true);
                                break;
                                }
                        }
                if (it==facts_.end()) //no reference found
                        unref_nolots[base]=n_tab->Fields->Fields[2]->AsString; //possible subtypes
                }
        n_tab->Next();
   }
```

```
//find references for possible subtype nolots
ss_iter is=unref_nolots.begin();
for (is;is!=unref_nolots.end();++is) {
	MyString parent((*is).second);
	if (!parent.IsEmpty()) {
		String raw_parent=parent.unwrap();
		//find primary reference for subtype Nolot
		siter ip=primrefs_.find(raw_parent);
		if (ip!=primrefs_.end())
			primrefs_.insert(make_pair((*is).first,(*ip).second));
		}
	}
}

void
RefReader::insert_ref(String& base_nolot,String& ref,bool unary)
{
TTable *l_tab = Form1->Table2; //table of Lot's
ss_pair checked_ref;
if (unary) {
	//check if explicit reference is lexical
	l_tab->SetKey();
	l_tab->Fields->Fields[0]->AsString = ref;
	if(l_tab->GotoKey()) {
		checked_ref = make_pair(ref,String("LOT"));
		this->mark_lot_as_used(ref);
		}
	//explicit reference must be non-lexical
	else checked_ref = make_pair(ref,String("NOLOT"));
	}
else checked_ref = make_pair(ref,String("COMPOSED"));
primrefs_.insert(make_pair(base_nolot,checked_ref));
}

void
RefReader::insert_subtype(String& supertype,String subtype)
{
svs_iter it=supertypes_.find(supertype);
if (it!=supertypes_.end())
	(*it).second.push_back(subtype);
else {
	vector<String> v;
	v.push_back(subtype);
	supertypes_.insert(make_pair(supertype,v));
	}
}

void
RefReader::mark_lot_as_used(String& lot)
{
sv_vector::iterator it;
bool deletable=true;
//check if lot is still involved in any fact type
for (it=facts_.begin();it!=facts_.end();++it)
	if ((*it)[0]==lot||(*it)[1]==lot) {
		deletable=false;
		break;
		}
//delete lot from memory
if (deletable) {
	for(it=lots_.begin();it!=lots_.end();++it)
		if ((*it)[0]==lot) break;
	if (it!=lots_.end()) lots_.erase(it);
	}
}

RefReader::~RefReader()
{
}
```

```
RefReader::iterator
RefReader::begin() const
{
return primrefs_.begin();
}

RefReader::iterator
RefReader::end() const
{
return primrefs_.end();
}

String
RefReader::find_type(String object_name)
{
TTable *n_tab = Form1->Table1; //table of Nolot's
TTable *l_tab = Form1->Table2; //table of Lot's
String type("");
//Nolot ?
n_tab->SetKey();
n_tab->Fields->Fields[0]->AsString=object_name;
if (n_tab->GotoKey()) type="NOLOT";
//Lot?
else {
        l_tab->SetKey();
        l_tab->Fields->Fields[0]->AsString=object_name;
        if (l_tab->GotoKey()) type="LOT";
        else type="NESTED";
        }
return type;
}

String
RefReader::find(String nolot)
{
String s("");
siter ir;
for (ir=primrefs_.begin();ir!=primrefs_.end();++ir)
        if ((*ir).first==nolot) s=((*ir).second.first);
return s;
}

void
RefReader::replace(String nolot,String dummy_ref)
{
siter ir;
for (ir=primrefs_.begin();ir!=primrefs_.end();++ir)
        if ((*ir).first==nolot) (*ir).second=make_pair(dummy_ref,String("DUMMY"));
}

ss_pair
RefReader::find_nested_objects(String fact_name)
{
TTable *f_tab = Form1->Table3; //table of facts
f_tab->First();
ss_pair ot;
while(!f_tab->Eof) {
        if (f_tab->Fields->Fields[4]->AsString==fact_name) {
                ot.first=f_tab->Fields->Fields[0]->AsString;
                ot.second=f_tab->Fields->Fields[1]->AsString;
                break;
                }
        f_tab->Next();
        }
return ot;
}

sv_vector&
RefReader::get_unmarked_lots()
{
return lots_;
}
```

```
sv_vector&
RefReader::get_unmarked_facts()
{
return facts_;
}

svs_map&
RefReader::get_inheritance()
{
return supertypes_;
}

bool
RefReader::empty() const
{
return (primrefs_.size()==0);
}
```

DtdBuilder.h

```
#ifndef DTDBUILDER_H
#define DTDBUILDER_H

#include <String.h>
#include "DtdEncoder.h"
#include "Refreader.h"

class DtdBuilder
{
public:
        DtdBuilder(DtdEncoder& d);
        ~DtdBuilder();

        void write(String& file_name,String& root,String& xml_version);
private:
        String  dtd_;
        void    build_xml_element(String name,String attributes,String content_model);
};

#endif
```

DtdBuilder.cpp

```
#include <vcl.h>
#include <fstream>
#include <vector.h>
#include <map.h>
#include "DtdBuilder.h"
#include "Main.h"

#pragma hdrstop

typedef vector<String>              s_vector;
typedef pair<String,String>         ss_pair;
typedef map<String,ss_pair>         spss_map;
typedef spss_map::iterator          spss_iter;
typedef map<String,s_vector>        svs_map;
typedef svs_map::iterator           svs_iter;
```

```
DtdBuilder::DtdBuilder(DtdEncoder& d)
{
RefReader& r_=d.get_refreader();
spss_map& elements=d.get_elements();
svs_map& super_types=r_.get_inheritance();
svs_iter it;
spss_iter is,im;
//supertype elements are built first
for (it=super_types.begin();it!=super_types.end();++it) {
        is=elements.find((*it).first);
        if (is!=elements.end()) {
                this->build_xml_element((*it).first,(*is).second.first,(*is).second.second);
                elements.erase(is);
                }
        }
//build elements based on nolot's
for (is=elements.begin();is!=elements.end();++is) {
        if (r_.find_type((*is).first)=="NOLOT")
                this->build_xml_element((*is).first,(*is).second.first,(*is).second.second);
        }
//build elements based on lot's
for (is=elements.begin();is!=elements.end();++is) {
        if (r_.find_type((*is).first)=="LOT")
                this->build_xml_element((*is).first,(*is).second.first,(*is).second.second);
        }
//build other elements
for (is=elements.begin();is!=elements.end();++is) {
        String type=r_.find_type((*is).first);
        if (type!="NOLOT"&&type!="LOT")
                this->build_xml_element((*is).first,(*is).second.first,(*is).second.second);
        }
}

DtdBuilder::~DtdBuilder()
{
}

void
DtdBuilder::build_xml_element(String name,String attributes,String content_model)
{
String empty_content("EMPTY"),element_core,attributes_core;
if (content_model.IsEmpty()) element_core=name+" "+empty_content;
else element_core=name+" ("+content_model+")";
//add XML element to DTD
dtd_+="<!ELEMENT "+element_core+">\n";
if(!attributes.IsEmpty()) dtd_+="<!ATTLIST "+name+" "+attributes+">\n\n";
else dtd_+="\n";
}

void
DtdBuilder::write(String& file_name,String& root,String& xml_version)
{
if (file_name.SubString(file_name.Length()-3,4) != ".dtd") file_name += ".dtd";
String version_header("<?xml version=\""+xml_version+"\"?>\n");
String doctype_header("<!DOCTYPE "+root+" [\n\n");
String dtd_footer("\n]>");
ifstream ifa(file_name.c_str());
if (ifa) ifa.clear(); //existing dtd's are always overwritten !!!
ofstream ofa(file_name.c_str(),(ios::out));
if (ofa) {
        ofa << (version_header+doctype_header+dtd_+dtd_footer).c_str();
        Form1->ListBox1->Items->Insert(0,"DTD \""+file_name+"\" successfully built.");
        }
else Form1->ListBox1->Items->Insert(0,"Could not write to DTD \""+file_name+"\"");
}
```

DtdEncoder.h

```
#ifndef DTDENCODER_H
#define DTDENCODER_H

#include <String>
#include <map.h>
#include <vector.h>
#include "RefReader.h"

typedef vector<String>              s_vector;
typedef s_vector::iterator          s_iter;
typedef vector<s_vector>            sv_vector;
typedef sv_vector::iterator         sv_iter;
typedef map<String,s_vector>        svs_map;
typedef svs_map::iterator           svs_iter;
typedef map<String,String>          ss_map;
typedef pair<String,String>         ss_pair;
typedef map<String,ss_pair>         spss_map;
typedef spss_map::iterator          spss_iter;

enum _target_loc_ {_first_,_second_};

class DtdEncoder
{
public:
        DtdEncoder(RefReader& r);
        ~DtdEncoder();

        void            add_attribute(String element,String name,String& type,String& req);
        void            add_content_model(String element,String new_cm);
        spss_map&       get_elements();
        RefReader&      get_refreader();

private:
        spss_map        elements_;
        RefReader&      r_;

        String    make_total_exclusive(s_vector& cm);
        String    add_constraints(String role_name,String ident,String mand,_target_loc_ rl);
        void      add_fact_nolot_nolot(s_vector fact,String ot1_ref,String ot2_ref);
        void      add_fact_lot_nolot(s_vector fact,_target_loc_ rl);
        void      add_nested_lot_nolot(String fact_name,String ot1,String ot2,String nolot_ref,_target_loc_ rl);
        void      add_nested_nolot_nolot(s_vector fact,String nolot1,String nolot2);

};

#endif
```

DtdEncoder.cpp

```
#include <vcl.h>
#include "DtdEncoder.h"

#pragma hdrstop

DtdEncoder::DtdEncoder(RefReader& r)
:r_(r)
{
//search for primary references for all nolot's
r_.find_primary_refs();
//make elements, attributes and content models for all nolot's
RefReader::iterator it;
String id("ID"),idref("IDREF"),req("#REQUIRED"),dummy_ref("Ref");
```

```
for (it=r_.begin();it!=r_.end();++it) {
	//lexical unary primary reference
	if ((*it).second.second=="LOT")
		this->add_attribute((*it).first,(*it).second.first,id,req);
	else if ((*it).second.second=="NOLOT")
		this->add_attribute((*it).first,(*it).second.first,idref,req);
	else if ((*it).second.second=="COMPOSED") {
		this->add_attribute((*it).first,dummy_ref,id,req);
		this->add_content_model((*it).first,(*it).second.first);
		//replace composed reference in memory
		r_.replace((*it).first,dummy_ref);
		}
	}
//complete content model of supertype nolot's
//this demo tool assumes totality and exclusion constraints on subtypes
//cfr. RULE 7
svs_map super=r_.get_inheritance();
svs_iter is;
for (is=super.begin();is!=super.end();++is) {
	s_vector sub=(*is).second;
	this->add_content_model((*is).first,make_total_exclusive(sub));
	}
//define unmarked lot's as XML elements of type #PCDATA
//cfr. RULE 8
sv_vector lt=r_.get_unmarked_lots();
sv_iter iv;
for (iv=lt.begin();iv!=lt.end();++iv)
	this->add_content_model((*iv)[0],String("#PCDATA"));
//fact types
//cfr. RULES 9,10,11,12,13,14
sv_vector ft=r_.get_unmarked_facts();
for (iv=ft.begin();iv!=ft.end();++iv) {
	s_vector fact=(*iv);
	String ot1_type(r_.find_type(fact[0])),ot2_type(r_.find_type(fact[1])),ident(fact[5]),mand(fact[6]);
	//unnested fact types
	if (ot1_type!="NESTED"&&ot2_type!="NESTED") {
		//fact type between unnested nolot's
		if (ot1_type=="NOLOT"&&ot2_type=="NOLOT") {
			String ot1_ref=r_.find(fact[0]),ot2_ref=r_.find(fact[1]);
			this->add_fact_nolot_nolot(fact,ot1_ref,ot2_ref);
			}
		//fact type between unnested lot and nolot
		else if(ot1_type=="LOT"&&ot2_type=="NOLOT")
			this->add_fact_lot_nolot(fact,_first_);
		//fact type between unnested nolot and lot
		else if(ot1_type=="NOLOT"&&ot2_type=="LOT")
			this->add_fact_lot_nolot(fact,_second_);
		}
	//both object types are nested
	else if (ot1_type=="NESTED"&&ot2_type=="NESTED") {
		ss_pair ot1=r_.find_nested_objects(fact[0]),ot2=r_.find_nested_objects(fact[1]);
		String not11_type(r_.find_type(ot1.first)),not12_type(r_.find_type(ot1.second));
		String not21_type(r_.find_type(ot2.first)),not22_type(r_.find_type(ot2.second));
		//add nested objects
		if (not11_type=="NOLOT"&&not12_type=="NOLOT")
			this->add_nested_nolot_nolot(fact,ot1.first,ot1.second);
		else if (not11_type=="LOT"&&not12_type=="NOLOT")
			this->add_nested_lot_nolot(fact[0],ot1.first,ot1.second,r_.find(ot1.second),_first_);
		else if (not11_type=="NOLOT"&&not12_type=="LOT")
			this->add_nested_lot_nolot(fact[0],ot1.first,ot1.second,r_.find(ot1.first),_second_);
		if (not21_type=="NOLOT"&&not22_type=="NOLOT")
			this->add_nested_nolot_nolot(fact,ot2.first,ot2.second);
		else if (not21_type=="LOT"&&not22_type=="NOLOT")
			this->add_nested_lot_nolot(fact[1],ot1.first,ot1.second,r_.find(ot2.second),_first_);
		else if (not21_type=="NOLOT"&&not22_type=="LOT")
			this->add_nested_lot_nolot(fact[1],ot1.first,ot1.second,r_.find(ot2.first),_second_);
		//add fact type between artificial nolots
		String art_ref1(fact[0]+"_ID"),art_ref2(fact[1]+"_ID");
		this->add_fact_nolot_nolot(fact,art_ref1,art_ref2);
		}
```

```
	else if (ot1_type=="NESTED") {
		ss_pair ot1=r_.find_nested_objects(fact[0]);
		String not11_type(r_.find_type(ot1.first)),not12_type(r_.find_type(ot1.second));
		//add nested object
		if (not11_type=="NOLOT"&&not12_type=="NOLOT")
			this->add_nested_nolot_nolot(fact,ot1.first,ot1.second);
		else if (not11_type=="LOT"&&not12_type=="NOLOT")
			this->add_nested_lot_nolot(fact[0],ot1.first,ot1.second,r_.find(ot1.second),_first_);
		else if (not11_type=="NOLOT"&&not12_type=="LOT")
			this->add_nested_lot_nolot(fact[0],ot1.first,ot1.second,r_.find(ot1.first),_second_);
		//add fact type with one artificial nolot
		if (ot2_type=="NOLOT") {
			String art_ref(fact[0]+"_ID");
			this->add_fact_nolot_nolot(fact,art_ref,r_.find(fact[1]));
			}
		else this->add_fact_lot_nolot(fact,_second_);
		}
	else if (ot2_type=="NESTED") {
		ss_pair ot2=r_.find_nested_objects(fact[1]);
		String not21_type(r_.find_type(ot2.first)),not22_type(r_.find_type(ot2.second));
		//add nested object
		if (not21_type=="NOLOT"&&not22_type=="NOLOT")
			this->add_nested_nolot_nolot(fact,ot2.first,ot2.second);
		else if (not21_type=="LOT"&&not22_type=="NOLOT")
			this->add_nested_lot_nolot(fact[1],ot2.first,ot2.second,r_.find(ot2.second),_first_);
		else if (not21_type=="NOLOT"&&not22_type=="LOT")
			this->add_nested_lot_nolot(fact[1],ot2.first,ot2.second,r_.find(ot2.first),_second_);
		//add fact type with one artificial nolot
		if (ot1_type=="NOLOT") {
			String art_ref(fact[1]+"_ID");
			this->add_fact_nolot_nolot(fact,r_.find(fact[0]),dummy_ref);
			}
		else this->add_fact_lot_nolot(fact,_first_);
		}
	}
}

DtdEncoder::~DtdEncoder()
{
}

void
DtdEncoder::add_nested_lot_nolot(String fact_name,String ot1,String ot2,String nolot_ref,_target_loc_ rl)
{
String id("ID"),idref("IDREF"),req("#REQUIRED");
//make XML element based on the name of the nested fact type
this->add_attribute(fact_name,String("Ref"),id,req); //dummy name
this->add_attribute(fact_name,nolot_ref,idref,req);
//adapt content model of fact type
switch (rl) {
	case _first_:
		this->add_content_model(fact_name,ot2);
		break;
	case _second_:
		this->add_content_model(fact_name,ot1);
		break;
	}
}

void
DtdEncoder::add_nested_nolot_nolot(s_vector fact,String nolot1,String nolot2)
{
String id("ID"),req("#REQUIRED"),dummy_ref("Ref");
//make XML elements based on the name of the nested fact types
this->add_attribute(fact[0],dummy_ref,id,req);
//adapt content model of fact type
this->add_content_model(fact[0],nolot1);
this->add_content_model(fact[0],nolot2);
}
```

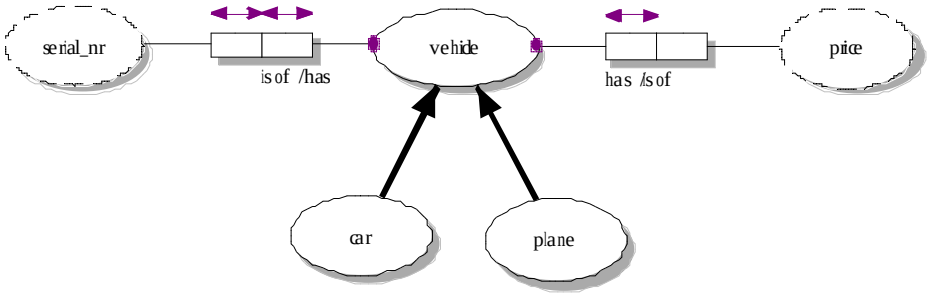
```
void
DtdEncoder::add_fact_lot_nolot(s_vector fact,_target_loc_ rl)
{
String ident(fact[5]),mand(fact[6]);
switch (rl) {
        //first object type is a lot
        case _first_:
                //make XML element based on role played by lot
                this->add_content_model(fact[3],fact[0]);
                //adapt content model of involved nolot
                this->add_content_model(fact[1],add_constraints(fact[3],ident,mand,_second_));
                break;
        //second object is a lot
        case _second_:
                //make XML element based on role played by lot
                this->add_content_model(fact[2],fact[1]);
                //adapt content model of involved nolot
                this->add_content_model(fact[0],add_constraints(fact[2],ident,mand,_first_));
                break;
        }
}

void
DtdEncoder::add_fact_nolot_nolot(s_vector fact,String ot1_ref,String ot2_ref)
{
String idref("IDREF"),req("#REQUIRED"),ident(fact[5]),mand(fact[6]);
//make XML elements based on the rolenames
//artificial nolot's have no reference in the tables
this->add_attribute(fact[2],ot2_ref,idref,req);
this->add_attribute(fact[3],ot1_ref,idref,req);
//adapt content model of involved ot's
this->add_content_model(fact[0],add_constraints(fact[2],ident,mand,_first_));
this->add_content_model(fact[1],add_constraints(fact[3],ident,mand,_second_));
}
String
DtdEncoder::add_constraints(String role_name,String ident,String mand,_target_loc_ rl)
{
String s("");
switch (rl) {
        case _first_:
                if (ident=="One-to-Many"||ident=="One-to-One") { //unique
                        if (mand=="Yes-Yes"||mand=="Yes-No") //mandatory
                                s=role_name;
                        else s=role_name+'?'; //non-mandatory
                        }
                else if (ident=="Many-to-One"||ident=="Many-to-Many") { //non-unique
                        if (mand=="Yes-Yes"||mand=="Yes-No") //mandatory
                                s=role_name+'+';
                        else s=role_name+'*'; //non-mandatory
                        }
                break;
        case _second_:
                if (ident=="Many-to-One"||ident=="One-to-One") { //unique
                        if (mand=="Yes-Yes"||mand=="No-Yes") //mandatory
                                s=role_name;
                        else s=role_name+'?'; //non-mandatory
                        }
                else if (ident=="One-to-Many"||ident=="Many-to-Many") { //non-unique
                        if (mand=="Yes-Yes"||mand=="No-Yes") //mandatory
                                s=role_name+'+';
                        else s=role_name+'*'; //non-mandatory
                        }
                break;
        default: s=role_name;
        }
return s;
}
```

```
void
DtdEncoder::add_attribute(String element,String name,String& type,String& req)
{
String new_attr(name+" "+type+" "+req);
spss_iter it=elements_.find(element);
if(it!=elements_.end()) {
        String old_attr((*it).second.first);
        //element already has an attribute
        if (!old_attr.IsEmpty())
                (*it).second.first="\n\t"+old_attr+"\n\t"+new_attr;
        }
else elements_[element]=make_pair(new_attr,"");
}

void
DtdEncoder::add_content_model(String element,String new_cm)
{
spss_iter it=elements_.find(element);
if(it!=elements_.end()) {
        String target_cm((*it).second.second);
        if (target_cm.IsEmpty()) (*it).second.second=new_cm;
        else (*it).second.second=target_cm+","+new_cm;
        }
else elements_[element]=make_pair("",new_cm);
}

String
DtdEncoder::make_total_exclusive(s_vector& cm)
{
s_iter it;
String s("");
for (it=cm.begin();it!=cm.end();++it)
        s+=(*it)+'|';
//remove last '|'
return s="("+s.SubString(0,s.Length()-1)+")";
}

spss_map&
DtdEncoder::get_elements()
{
return elements_;
}

RefReader&
DtdEncoder::get_refreader()
{
return r_;
}
```

Appendix 5:

Examples of the BibTeX-based file format for TransORM

- Example 1



@NOLOT{name = vehicle, reference = [serial_nr], parents =[]}

@NOLOT{name = car, reference = [serial_nr], parents = [vehicle]}

@NOLOT{name = plane, reference = [serial_nr], parent = [vehicle]}

@LOT{name = serial_nr, category = unit}

@LOT{name = price, category = unit}

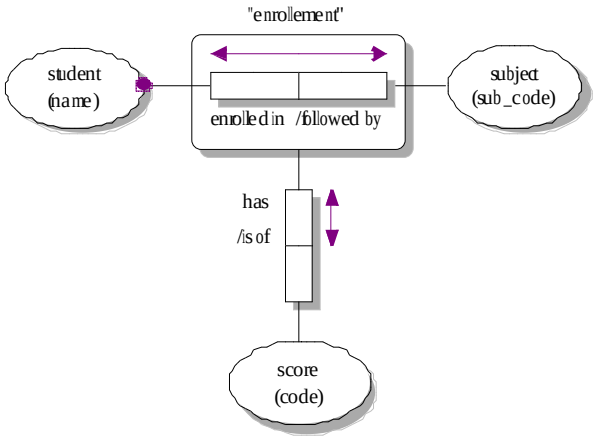
@FACT{ objects = [vehicle,serial_nr],
rolenames = [has,is of],
factname = [],
uniqueness = oto,
mandatory = YN}

@FACT{ objects = [vehicle,price],
rolenames = [has,is of],
factname = [],
uniqueness = otm,
mandatory = YN}

@FACT{ objects = [car,serial_nr],
rolenames = [referenced by,reference of],
factname = [],
uniqueness = oto,
mandatory = YN}

@FACT{ objects = [plane,serial_nr],
rolenames = [referenced by,reference of],
factname = [],
uniqueness = oto,
mandatory = YN}

- Example 2



@NOLOT{name = student, reference = [name], parents = []}

@NOLOT{name = subject, reference = [sub_code], parents =[]}

@NOLOT{name = score, reference = [code], parents = []}

@LOT{name = name, category = label}

@LOT{name = sub_code, category = unit}

@LOT{name = code, category = unit}

@FACT{ objects = [student,name],
rolenames = [referenced by,reference of],
factname = [],
uniqueness = oto,
mandatory = YN}

@FACT{ objects = [subject,sub_code],
rolenames = [referenced by,reference of],
factname = [],
uniqueness = oto,
mandatory = YN}

@FACT{ objects = [score,code],
rolenames = [referenced by,reference of],
factname = [],
uniqueness = oto,
mandatory = YN}

@FACT{ objects = [student,subject],
rolenames = [enrolled in,followed by],
factname = [enrollement],
uniqueness = mtm,
mandatory = YN}

@FACT{ objects = [enrollement,score],
rolenames = [has,is of],
factname = [],
uniqueness = otm,
mandatory = NN}