XDO2: AN XML DEDUCTIVE OBJECT-ORIENTED QUERY LANGUAGE

ZHANG WEI

NATIONAL UNIVERSITY OF SINGAPORE

2004

XDO2: AN XML DEDUCTIVE OBJECT-ORIENTED QUERY LANGUAGE

ZHANG WEI
(*B.Comp.(Hons.) NUS*)

A THESIS SUBMITTED
FOR THE DEGREE OF MASTER OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2004

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Ling Tok Wang, for his invaluable guidance and advice throughout my whole research work. He has been extremely helpful and has always provided invaluable ideas on the research topic. I would like to thank him also for his kindness, patience, and ingenuity in guiding me to solve the problems. His priceless remarks, suggestions and support always encouraged me to try my best to make the project better.

I would also like to thank Associate Professor Gillian Dobbie for her advice and effort in my research work. She has been extremely helpful and always provided guidance in my research work.

Finally, I would like to thank Mr. Chen Zhuo for his advice and help in my research work and my lab fellows, Ni Wei, He Qi, Li Changqing and Jiao Enhua for their generous suggestions and help in my research, and for the pleasant and friendly environment of the database research lab.

Table of Contents

List of Figures

List of Tables

Summary

In the past decade, researchers have combined deductive and object-oriented features to produce systems that are powerful and have excellent modeling capabilities. Many deductive and object-oriented features have been investigated, such as deductive rules, negation, inheritance and multiple inheritance with conflict handling.

XML is fast emerging as the dominant standard for data representation and exchange in the web. How to query XML documents to extract and restructure the data is still an important issue in XML research. Currently, XQuery based on XPath is the standard XML query language from W3C. However, it has some limitations and XTree has recently been proposed. Queries written in XTree are more compact, more convenient to write and easier to understand than queries written in XPath.

In this thesis, we propose a novel XML query language XDO2 which is based on XTree and has deductive database features such as *deductive rules* and *negation*, and object-oriented features such as *inheritance* and *methods*. The major contributions of the XDO2 query language are:

1. Negation is supported in the XDO2 language with semantics similar to the not-predicate instead of the conventional logic negation which

negates the boolean predicate and is used in XQuery. A consequence of this decision is that XDO2 is able to support nested negation and negation of sub-trees in a more compact form.

2. Methods that deduce new properties are implemented as deductive rules. XDO2 can use the new properties directly. The presence of recursive deductive rules makes recursive querying possible.

3. Schema querying is made possible with a special term *stru : value* to explicitly distinguish the element tag (attribute name) from the element value (attribute value). *Stru* binds to the element tag (attribute name) and *value* binds to the element value (attribute value). Unlike in XQuery, the name and value pairs are bound to the variables together.

4. Inheritance enables a subclass object to inherit all the attributes, subelements and methods from its superclass objects. These inherited properties can be directly used in querying.

5. Features such as the binding of multiple variables in one expression, compact return format and explicit multi-valued variables are supported in the XDO2 language naturally due to the influence of XTree.

In summary, we have developed a more compact, convenient to use, and powerful XML query language with deductive rules, not-predicate negation, and the support of some object-oriented features.

In addition, a database example is presented to motivate the discussion of our XDO2 language. The formal treatment of the language syntax and semantics are presented. We also present some extensions on XML Schema definitions in the appendix.

Chapter 1

Introduction

In the 1980's, the object-oriented paradigm was introduced and became very popular because it can naturally model the real world objects. Some object-oriented programming languages were proposed, developed and widely used, such as Java, C++ and C#. Many object-oriented data models, such as ONTOS [38], O2 [19], Orion [24], IRIS [18] and ObjectStore [25] have been proposed. Based on these, a large number of object-oriented features have been proposed. These features include object identity, complex object, typing, class inheritance, overriding, blocking, multiple inheritances with conflict handling mechanism, method encapsulation, method overloading, late binding, and polymorphism.

Deductive databases are an extension of relational databases that support a more powerful query language. In deductive databases, the most important feature is the introduction of deductive rules to derive new information. Especially the presence of recursive deductive rules makes recursive querying possible. Another important feature is negation which enables more meaningful queries.

In the past decade, a large number of deductive object-oriented database systems have been proposed, such as O-logic [34], C-logic [9], F-logic [23], IQL [1], LOGRES [8], LLO [33], $Datalog^{meth}$ [2], CORAL++ [39], ROCK&ROLL [4], Gulog [16], ROL [30], Datalog++ [21], and DO2 [27]. The objective of deductive object-oriented databases is to combine the best of the deductive and object-oriented approaches, namely to combine the logical foundation of the deductive approach with the modelling capabilities of the object-oriented approach.

In the late 1990's, XML which provides a way to represent typical semi-structured data is fast emerging as the dominant standard for data representation and exchange in the web. The flexibility of XML means it is widely used for data exchange. However, the world is full of compromise, and due to the flexibility of XML the relational data model is not suitable for storing XML data since the structure of XML is not fixed. Many researchers are researching this new kind of data from many perspectives. How to store XML data and how to query XML data efficiently are still the important fields in the new area of database research and development. In our thesis, we will not talk about how to store the XML data, but we will concentrate on how to query the XML data, how to make the query more simple, compact while still declarative and powerful. We assume the XML data is a tree structure data. Many query languages have been proposed in the past few years, such as XPath [13], XQuery [6], Lorel [3], XQL [36], XSLT [12], a rule-based query language [11], a declarative XML query language [29] and XTree [10]. Among these query languages, XQuery [6] developed by W3C has become a standard and widely been accepted by the XML database research community. XTree

[10] has been proposed as an XML query language to be more compact, more convenient to write and understand than XPath [13]. However, these XML query languages currently can not support the deductive and object-oriented features in the XML database system. How to apply these deductive object-oriented features into current XML query languages is the main focus of our thesis.

In this thesis, we propose a novel XML query language XDO2 which is based on XTree and has deductive object-oriented features. We present an XDO2 database example to give users an overview of what the XML deductive object-oriented database looks like and motivate the discussion of the XDO2 language. Some important features are presented in the example, such as deductive rules and inheritance. The XDO2 language features are presented in a more systematic and detailed way to include the features coming from XTree, such as multiple variables in one expression, compact return format, aggregation functions, and naturally separating structure from value. It also includes negation and recursion features coming from deductive databases and the features coming from object-oriented databases. The syntax and the semantics of the XDO2 language are formally defined. In addition, we also investigate the definition of deductive rules, the definition of relationship type as in the ORA-SS [17] model and the definition of the superclass attribute in XML Schema [5, 20, 40].

The major contributions of the XDO2 query language are:

1. Negation is supported in the XDO2 language with semantics similar to the not-predicate instead of the conventional logic negation which negates the boolean predicate and is used in XQuery. A consequence

of this decision is that XDO2 is able to support nested negation and negation of sub-trees in a more compact form.

2. Methods that deduce new properties are implemented as deductive rules. XDO2 can use the new properties directly. The presence of recursive deductive rules makes recursive querying possible.

3. Schema querying is made possible with a special term *stru : value* to explicitly distinguish the element tag (attribute name) from the element value (attribute value). *Stru* binds to the element tag (attribute name) and *value* binds to the element value (attribute value). Unlike in XQuery, the name and value pairs are bound to the variables together.

4. Inheritance enables a subclass object to inherit all the attributes, subelements and methods from its superclass objects. These inherited properties can be directly used in querying.

5. Features such as the binding of multiple variables in one expression, compact return format and explicit multi-valued variables are supported in the XDO2 language naturally due to the influence of XTree.

The rest of the thesis is organized as follows. We introduce the preliminary works including XTree, deductive databases, and object-oriented databases in chapter 2. We introduce the XML deductive object-oriented database using an example in chapter 3. Chapter 4 describes the features of the XDO2 language. Chapter 5 defines the syntax of the XDO2 language and chapter 6 defines the semantics of the XDO2 language. In chapter 7, we compare the XDO2 language with related works. Finally, chapter 8 sum-

marizes this thesis and points out some future research directions. In the appendix, some extensions of the XML Schema are presented.

Chapter 2

Preliminaries

XML is becoming prevalent in data representation and data exchange on the Internet. Many XML query languages have been proposed and XPath [13] which is a linear navigational path to the target XML node set is used in some of these query languages. However, XPath has some limitations and XTree [10] has been proposed to resolve these limitations. XTree is designed to have a tree structure instead of a path structure as in XPath. The advantages of XTree as an XML path language over XPath will be explained in more detail in section 2.1.

In relational databases, recursive queries are not supported in relational algebra or relational calculus. However, in deductive databases, recursive queries are supported naturally. Deductive databases have an extensional database and an intentional database. The extensional database are those data facts which correspond to the relational database tuples. The intentional database is composed of a list of rules so that some new facts can be deduced from the current data facts. If the rules are defined recursively, then recursive queries are supported naturally. Another important issue in

deductive database is negation. It enables more meaningful queries, but it complicates the query interpretation and evaluation. These deductive features are explained in more detail in section 2.2.

The relational database systems support only a small, fixed collection of data types (e.g., integers, dates, strings), which can not handle complex kinds of data. The object-oriented database was introduced in the 1980's and became very popular because it can naturally model the real world objects in a human's mind and support complex data types which are needed in some applications. The features in object-oriented paradigm include object identity, complex object and typing, class, inheritance, overriding and blocking, multiple inheritance and conflict handling, method encapsulation, overloading and late binding, and polymorphism. Section 2.3 describes these features in more detail.

2.1 XTree

XTree [10] has been proposed as an XML path language. The major contribution of XTree is to use square bracket [ ] to group the same level attributes and elements together so that the query languages based on XTree have a tree structure instead of a path structure in XPath [13]. In the following, we compare XTree with XPath and XTreeQuery [10] (an XML query language based on XTree) with XQuery [6] (an XML query language based on XPath) using some examples.

Example 2.1. Find the year and title of each book, and its authors' last name and first name.

XTree expression:

$/bib/book/[@year \rightarrow \$y, title \rightarrow \$t, author/[last \rightarrow \$last, first \rightarrow \$first]]$

XPath expressions:

$book in /bib/book, $y in $book/@year, $t in $book/title, $author in
$book/author, $last in $author/last, $first in $author/first

As we can see from the above, one XTree expression corresponds to the six XPath expressions although they express the same meaning. The XTree expression is much more simple and compact using the square bracket [ ] to group the same level attributes and elements. We also noticed the XTree expression has only four variables defined which are the interested information while there are six variables in XPath expressions. The extra variables $book and $author are necessary to keep the correlation between the variables.

Example 2.2. List the titles and publishers of books which are published after 2000.

XTreeQuery expression:

query /bib/book/[@year → $y, title → $t, publisher → $p]
where $y > 2000
construct /result/recentbook/[title ← $t, publisher ← $p]

XQuery expression:

for $book in /bib/book, $y in $book/@year, $t in $book/title,
$p in $book/publisher
where $y > 2000
return <result><recentbook>{$t}{$p}</recentbook></result>

As we can see from the result construction part of both the XTreeQuery and XQuery, only one XTree expression is used for the query result format.

However, in XQuery, the XML element tags are mixed with the XPath expressions. Therefore, with the help of XTree, the result format is much more simple and compact instead of mixing the element tags with XPath expressions.

Example 2.3. List the title of the books that have more than 1 author.

XTreeQuery expression:

query /bib/book/[title → $t, author → {$a}]

where {$a}.count() > 1

construct /result/multiAuthorBook/title ← $t

XQuery expression:

for $book in /bib/book, $t in $book/title

let $a in $book/author

where count($a) > 1

return <result><multiAuthorBook>{$t}</multiAuthorBook></result>

In XQuery, there is no syntactic difference between single-valued variables and multi-valued variables, but the multi-valued variables are defined in the *let* clause. However, in XTree, the multi-valued variables are explicitly indicated by surrounding curly braces { }. Therefore, the *let* clause can be avoid in those query languages based on XTree. We also noticed that the object-oriented fashion built-in aggregate functions are supported in XTree, such as *{$a}.count()* in this example. However, in XQuery, the built-in aggregate functions are supported as functions, such as *count($a)*.

Example 2.4. Obtain some attribute with value 2000 in some book element.

XTreeQuery expression:

query /bib/book/@$attr → $value

where $value = 2000

construct $attr

XQuery expression:

*for $pair in /bib/book/@**

where string($pair) = "2000"

return local-name($pair)

As we can see from this example, in XTree, the XML attribute names (element tags) are separated from the values naturally. The left of symbol → binds to the structure while the right of the symbol → binds to the value. However, in XQuery, we have to use symbol "*" (all) since the structure is unknown and have to use two built-in functions *string()* and *local-name()* to get the values and attribute names (element tags) respectively.

As a summary, XTree is a generalization of XPath [13] and has the following advantages over XPath or XQuery [6].

1. In XPath, it is only possible to specify a linear path to the target XML node set. In the querying part of a query, such as an XQuery expression, one XPath expression can only bind one variable. However, XTree has a tree structure which is similar to the structure of an XML document. In the querying part of a query, such as an XTreeQuery expression, one XTree expression can bind multiple variables.

2. XPath cannot be used to define the return format. However, in the result format part of a query, one XTree expression can be used to define the result format. This effectively avoids the mixing of element

tags (attribute names) with XPath expressions and nested structure in the result format in XQuery.

3. In XTree expressions, multi-valued variables are explicitly indicated, and their values are uniquely determined. Some natural built-in aggregate functions are defined to manipulate multi-valued variables in an object-oriented fashion. However, in XPath, there is no difference between single-valued variables and multi-valued variables. In XQuery, it must use a *let* clause to define the multi-valued variables. The built-in functions are in functional fashion instead of object-oriented fashion in XQuery.

4. In XTree expression, the element tags (attribute names) are separated from the values naturally using the term structure $left \rightarrow right$. This will make the querying on the structure or schema more convenient. However, in XPath, the variables bound to the structure and value together. Therefore, symbol "*" (all) must be used and built-in functions are used to query the schema in XQuery.

Thus, although XPath and XTree have the same expressive power (i.e., anything that can be expressed by XTree can also be expressed by several XPaths), XTree is more compact and convenient to use than XPath, and queries based on XTree expressions are shorter in length and easier to write and comprehend. In short, XTree is designed to have a tree structure while XPath does not. For more details, please refer to [10].

2.2 Deductive Databases

Deductive databases are an extension of relational databases to support more powerful querying, such as recursive querying. Deductive databases consist of *extensional databases*, which are exactly the relational database relations and *intentional databases*, which are composed of a collection of *rules*. The rules defined in a query langauge called *Datalog* [41] are used to deduce new data tuples from the extensional databases. Datalog, which is a relational query language is inspired by *Prolog*, the well-known logic programming language and the notation of Datalog follows Prolog.

Example 2.5. Consider the following collection of rules.

ancestor(X, Y) :- parent(X, Y).

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

These are rules in Datalog and the first rule means if there is a tuple <X, Y> in a parent relation, then there must be a tuple <X, Y> in the ancestor relation. The second rule means if there is a tuple <X, Z> in a parent relation and a tuple <Z, Y> in an ancestor relation, then there must be a tuple <X, Y> in the ancestor relation.

The part to the right of the :- symbol is called the *body* of the rule, and the part to the left is called the *head* of the rule. All the variables start with an uppercase letter, such as X, Y, and Z and constants start with a lowercase letter.

Suppose there is a parent relation shown in Table 2.1. By applying the first rule, we can get the ancestor relation with exactly the same four tuples as in parent relation. Now we have four tuples in the parent relation and the four newly deduced tuples in ancestor relation.

Now we do not need to consider the first rule since it can not deduce new tuples because parent relation does not have any new tuples created. We can apply the second rule by considering the cross-product of parent relation and ancestor relation. Notice the repeated use of variable Z in both parent and ancestor relation in the rule. It means the two column values should be equal and in fact it specifies an equality join condition on parent and ancestor relation. After applying the second rule once, two newly deduced tuples <john, sandy>, <mary, lucy> are created in the ancestor relation.

Now we still do not need to consider the first rule since no new tuples are created in the parent relation. Notice it is not advisable to considering the cross-product of parent relation and the whole ancestor relation since parent relation does not change and many tuples in ancestor relation have already joined with the parent relation in the previous step. We only need to join the parent relation with the two new tuples in ancestor relation. Then we get one more tuple <john, lucy> in the ancestor relation.

Again, after applying the second rule with parent relation and the newly created tuple in ancestor relation, we can not get any more tuples in ancestor relation. And finally, the ancestor relation is created and shown in Figure 2.2.

Now suppose there is a query to query john's descendants as follows,

:- ancestor(john, X).

Since ancestor relation is created using the deductive rules, we can get john's descendants naturally. The result will be X = mary, X = ben, X = sandy, and X = lucy.

In this example, we have seen the most important feature of deductive

parent	child
john	mary
john	ben
mary	sandy
sandy	lucy

Table 2.1: Parent relation

ancestor	descendant
john	mary
john	ben
mary	sandy
sandy	lucy
john	sandy
mary	lucy
john	lucy

Table 2.2: Ancestor relation

databases. Rules can be used to deduce new tuples in a relation. Recursive rules recursively deduce new tuples and support recursive querying naturally. In the next section, the meaning or semantics of the deductive rules are defined.

2.2.1 Semantics of deductive rules

Given a set of deductive rules in deductive databases, there are two approaches to define the semantics. The first approach is called the *least model semantics*, which gives users a way to understand the program (deductive rules) without thinking about how the program is to be executed and the second approach is called the *least fixpoint semantics*, which gives a conceptual evaluation strategy to compute the desired relation instances.

Before defining the least model, we need to define the model first. A *model* is a collection of relation instances such that the relation instances satisfies all the rules in the sense that for each rule, after we replace the

variables by constants, if the tuples in the body are in the relation instances, then the tuples generated in the head are in the relation instances.

Observe that in the Example 2.5, the parent instance shown in Table 2.1 and the ancestor instance shown in Table 2.2 actually form a model of the deductive rules. This is because for the first rule, every tuple of the parent relation instance in the body, the tuple generated for the ancestor relation is also in the ancestor relation instance. Also for the second rule, every tuple of the parent relation instance joined with every tuple of the ancestor relation instance in the body, the tuple generated for the ancestor relation is also in the ancestor relation instance.

However, suppose we add one more tuple such as <john, dale> into the ancestor instance, the parent and the new ancestor instances satisfies the first rule trivially. They satisfies the second rule also since <john, dale> can not be used to join with parent instance to generate new tuple in ancestor relation. Therefore, the parent instance with the new ancestor instances (by added one more tuple) form a model too. In order to make the semantics unique, the *least model* is defined to be a model M such that for any other model M2 of the same rules, M is "minimum" in the sense that for every tuple in the instance of M, the tuple is also in the instance of M2.

The above definition only defines the conditions that the least model satisfies. It does not give an evaluation strategy on how to compute the least model. *Fixpoint* is defined to be an instance such that the deductive rules applied to the fixpoint instance returns the same fixpoint instance. *Least fixpoint* is defined that the instance is smaller than every other fixpoint similar to the definition of least model.

In example 2.5, we have roughly go through on how to get the least fixpoint step by step. Initially, the parent instance and the empty ancestor instance are not fixpoint since after the rules are applied to the input instance, the output ancestor instance (four new tuples) is not the same as the input ancestor instance (empty). Therefore, we must add the output tuples into ancestor instance to try to make it be a fixpoint. When we apply the rules to the new input instance again, we get two more ancestor tuples not in the input instance. Then we add the new output tuples into the input instances again and repeat the process until every tuple generated is already in the current instance. Therefore, intuitively, with this process is repeated, the fixpoint computed is the least fixpoint.

In fact, the least model and least fixpoint are identical. It has also shown that every Datalog program (deductive rules) has a least fixpoint and it can be computed by repeatedly applying the rules on the given relation instances.

Unfortunately, when set-difference (negation) is allowed in the body of the rule, there may no longer be a least model or a least fixpoint and we explain it in the next section.

2.2.2 Stratification

Generally, every Datalog program has a least fixpoint which can be computed by repeatedly applying the rules on the given relation instances. But with set-difference (negation), which is the logical negation allowed inside the body of the rule, there may not be a least fixpoint for the set of rules. In this case, a technique called *stratification* is used to resolve this problem. You can find some more details on [35].

Example 2.6. Consider the following set of rules.

solid (X) :- substance (X), not liquid (X), not gas (X).

liquid (X) :- substance (X), not solid (X), not gas (X).

gas (X) :- substance (X), not solid (X), not liquid (X).

The first rule defines a substance X is solid if it is not liquid and not gas. The second rule defines a substance X is liquid if it is not solid and not gas. And the third rule defines a substance X is gas if it is not solid and not liquid.

Assume there is only one tuple <book> in substance relation instance. If we apply the first rule first, <book> will be the newly generated tuple for the solid relation (solid, liquid and gas relation instances are initially empty). However, if we apply the second rule first, <book> will be in the liquid relation. <book> will be in the gas relation if we apply the third rule first. This program (deductive rules) has three fixpoints, none of which is smaller than the other two. Therefore, there is no least fixpoint in this program which involves negation.

A widely used solution to the problem caused by negation, or the use of not, is to impose some strata or layers to the relations and so called stratification. We say that a relation T *depends on* a relation S if some rule with T in the head contains S, or (recursively) contains a predicate that depends on S, in the body. We say that a relation T depends negatively on a relation S if some rule with T in the head contains *not* S, or (recursively) contains a predicate that depends negatively on S, in the body. For example, solid depends (negatively) on liquid, gas and recursively (negatively) on itself.

We classify the relations in the program into *strata* as follows. The rela-

tions that do not depend on any other tables are in stratum 0. In example 2.6, only the substance relation is in stratum 0. The relations in stratum 1 are those that depend only on relations in stratum 0 or stratum 1 and depend negatively only on relations in stratum 0. The relations in stratum i are those that do not appear in lower strata, depend only on relations in stratum i or lower strata, and depend negatively only on relations in lower strata. A program is *stratified* if and only if it can be classified into strata according to the above algorithm.

The example 2.6 is not stratified since solid, liquid and gas depend on each other, they must be in the same stratum. However, they depend negatively on the other two, violating the requirement that a relation can depend negatively only on relations in lower strata.

Example 2.7. Consider the following variant of the program,

solid2 (X) :- substance (X), static (X).

liquid2 (X) :- substance (X), flow (X), not solid2 (X).

gas2 (X) :- substance (X), not solid2 (X), not liquid2 (X).

This program is stratified. Liquid2 depends on solid2, and gas2 depends on solid2 and liquid2 but not vice versa. Substance, static and flow are in stratum 0, solid2 is in stratum 1, liquid2 is in stratum 2, and gas2 is in stratum 3.

A stratified program is evaluated stratum-by-stratum, starting with stratum 0. To evaluate a stratum, we compute the fixpoint of all rules defining relations that belong to this stratum. Therefore, when evaluating a stratum, any occurrence of *not* involves a relation from a lower stratum, which has already been evaluated. Intuitively, the requirement that programs be

stratified gives us a natural order for evaluating rules. When the rules are evaluated in this order, the result is a unique fixpoint which usually corresponds well to our intuitive reading of a stratified program.

In this section, we have covered the most basic features in deductive database. Deductive rules which can be used to deduce new tuples are talked about and the use of recursive rules for recursive querying is presented. The semantics of the rules are defined precisely using both least model and least fixpoint. We have also covered the negation feature, the problem it brings, and how to solve it. For more details, please refer to [41].

2.3 Object-Oriented Databases

The object-oriented paradigm became very popular in the 1980's. It has been applied in many areas, such as the programming langauges Java, C++, C#. Object-oriented databases support more complex object structure than the flat table in relational databases. Object identity, complex object and typing, class, inheritance, overriding and blocking, multiple inheritance and conflict handling, method encapsulation, overloading, late binding, and polymorphism are some main features in object-oriented databases.

2.3.1 Object identity

In object-oriented database systems, data objects have an *object identifier* (oid), which is some value that is unique in the database across time. The database management system is responsible for generating oids and ensuring that an oid identifies an object uniquely over its entire lifetime. Many de-

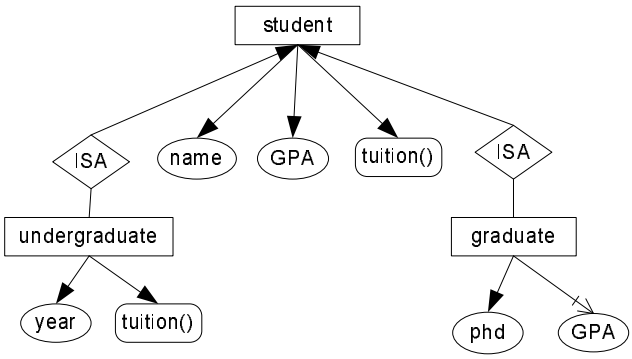

Figure 2.1: An inheritance diagram

ductive object-oriented database systems, including Florid [22], Rock & Roll [4], and Rol [30] implement object-identifier using OIDs.

2.3.2 Complex object and typing

In relational database systems, only a small, fixed collection of data types (e.g., integers, dates, strings) are supported. However, in the real world and in many application domains, much more complex kinds of data must be handled. In order to meet such applications' requirement, the database management systems must be able to support complex data types which are defined by the applications.

2.3.3 Class, inheritance, overriding and blocking

In the object-oriented paradigm, objects are defined in terms of *classes*. A *class* is the model, or pattern, from which an object is created. A class defines some properties, including attribute and behaviors (methods) such that all the objects defined by the class have the same properties.

Not only objects are defined by classes, object-oriented systems allow classes to be defined in terms of classes. As Figure 2.1 shows, *undergraduate* class and *graduate* class are all kinds of *student* class. In object-oriented terminology, *undergraduate* and *graduate* are all *subclasses* of the *student* class. Sometimes, we said *undergraduate ISA student* and *graduate ISA student*. Similarly, the *student* class is the *superclass* of *undergraduate* and *graduate*.

Each subclass can *inherit* all the properties from its superclass, such as attributes *name*, *GPA*, and method *tuition()* are inherited by the *undergraduate* and *graduate* subclasses. Besides these inherited properties, the subclasses can have their own properties, such as *year* in *undergraduate* and *phd* in *graduate*.

Subclasses can also *override* the properties from its superclass and provide specialized implementations for the properties, such as *undergraduate* class overrides the *tuition()* method so that the *tuition()* method has a different specialized interpretation in *undergraduate* class.

Subclasses can also *block* the properties that they do not want to inherit from their superclasses, such as *graduate* class blocks the *GPA* so that *GPA* will not be inherited from the *student* class. Notice we use an arrow with a dash to block a property as shown in Figure 2.1.

2.3.4 Multiple inheritance and conflict handling

In the previous section, we covered inheritance (the feature of inherit) and this is a good place to begin discussing multiple inheritance, which is one of the more powerful and challenging concepts in the object-oriented paradigm.

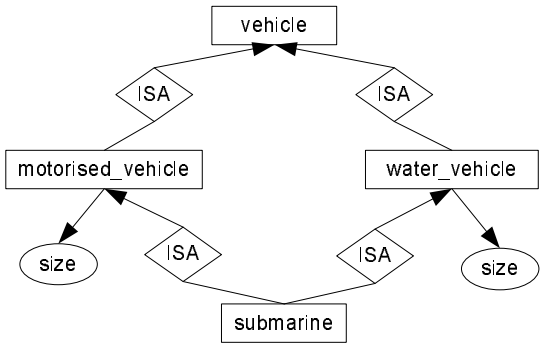

Figure 2.2: A multiple inheritance diagram

As the name implies, *multiple inheritance* allows a class to inherit properties from more than one class. This is a great idea which models the real world more naturally and there are many real-world examples of multiple inheritance. For example, parents are a good example of multiple inheritance. Each child has two parents and inherits the properties from his/her parents. In some object-oriented languages, such as C++, multiple inheritance is supported. However, multiple inheritance can significantly increase the complexity of a system and bring many problems, such as *multiple inheritance conflict*. Multiple inheritance conflict occurs when a class inherits several commonly named properties from its superclasses.

Figure 2.2 shows a multiple inheritance example. Class submarine has two superclasses: motorised_vehicle and water_vehicle which are subclasses of vehicle. As we can see, *size* is defined in both motorised_vehicle and water_vehicle. Thus, submarine does not know which *size* attribute it should inherit.

Many techniques are used to resolve such conflicts. Such as choosing the first in the list of superclass in ORION [24]. POSTGRES [37] do not allow

the creation of a subclass that inherits conflicting attributes. O2 [19] allows the explicit selection of the properties to inherit and emphasizes the path along which the property is to be inherited from. IRIS [18] uses the type information to resolve conflicts.

Many of these techniques examine neither the semantics of the properties involved in a conflict situation nor the reasons for the conflict. In [28], such conflicts are resolved by using a *Conflict Resolution Algorithm* considering the semantics of the properties and the reasons for the conflict. The techniques include *redesigning the schema, removing redundant ISA relationship, redefining an overload property, renaming properties, factoring attributes to a more general class, and explicitly selecting the desired property.*

Redesigning the schema is used when the schema design is poor or erroneous. For example, given a subclass, the intersection of its superclasses may be empty. Therefore, the subclass objects can not exist in the real world and the schema should be redesigned.

Since ISA relationship is transitive, it is possible that there exist redundant ISA relationship in the schema design. For example, if there is a ISA relationship between submarine and vehicle and the ISA relationship between submarine and water_vehicle is dropped. Suppose the vehicle class also defines the attribute *size*. In this case, we can *remove the redundant ISA relationship* between submarine and vehicle since submarine is a motorised_vehicle which is a vehicle. Therefore, multiple inheritance is changed to single inheritance and the conflict problem is solved.

Suppose in Figure 2.2, the submarine class also defines its own *size* attribute which *redefines* the *size* attribute from its superclasses. Therefore,

the submarine has its own *size* attribute and does not need to worry about which class the attribute *size* it should inherit from.

Given a subclass, the inherited properties of its superclasses may have the same name but different semantics. For example in Figure 2.2, if *size* in class motorised_vehicle means *weight* while the *size* in class water_vehicle means *capacity*, then *renaming* is recommended to resolve the conflict by changing the name of *size* in motorised_vehicle to *weight* and the name of *size* in water_vehicle to *capacity*.

When the conflicting inherited attributes have the same semantics, we can *factor* the attributes to a more general class. For example, the *size* attribute can be moved to the more general class vehicle. In this case, both motorised_vehicle and water_vehicle do not have *size* attribute and inherit it from the general class vehicle. The submarine can also inherit it from the general class vehicle without the conflict problem.

Another way to solve the same semantics conflicting attributes is by *explicitly selecting* the desired property. The submarine can explicitly select the class name of the *size* attribute it wants to inherit from.

2.3.5 Method encapsulation, overloading and late binding

In the object-oriented paradigm, methods are used to describe the behaviors of objects. *Method encapsulation* is defined by having methods defined within the class definitions instead of outside of class scope.

The method is identified by its name, the return type and its list of arguments. When two methods with the same name but different list of

arguments (including the difference of the number of arguments, the types of the arguments and the order of the types), then we say one method *overloads* another method.

A subclass may implement a method to override that in its superclass. Method resolution that determines which implementation is associated with a given method name and class at runtime, is known as *late binding*. Suppose Manager ISA Employee and *bonus()* is a method implemented in both Manager and Employee. When a *bonus()* message is sent to an instance of Employee who is also a manager, the *bonus()* method in Manager class is executed instead of *bonus()* method in Employee class if late binding is supported in the system. Method late binding is also known as dynamic binding since the method resolution is determined at runtime instead of compile time.

2.3.6 Polymorphism

Inheritance makes another key object-oriented concept *polymorphism* possible. In a programming language definition, it is used to express the fact that the same message (method name) can be sent to different objects and interpreted in different ways by each object. In this meaning, it is equivalent to the definition of method late binding. However, *polymorphism* has some other meanings. For example, motorcycle and car are subclasses of vehicle class. Then we can group the motorcycle objects and cars objects together in a set with type vehicle. In this way, polymorphism makes different kinds of objects organized together, and each object retains their individual properties.

Chapter 3

XDO2 Database Example

In this chapter, using an example, we demonstrate many of the features of the XDO2 language. We show an XDO2 database example to motivate the discussion of XDO2 query language. The database presented is *Person_Company_Employee*, which combines features from XML, deductive databases, and object-oriented databases.

In section 3.1, we present the database schema using the ORASS model [17]. The schema is extended to include the deductive rules and the object-oriented features such as class hierarchy relationships (relationships of class inheritance). We also briefly explain how to express deductive rules in our XDO2 database. In section 3.2, we present the XML database data, including the XML extensional data element facts, intentional data which are the deductive rules, and the class hierarchy relationships. An XDO2 query with its result is also presented. The syntax and semantics of the XDO2 language are presented in chapter 5 and 6.

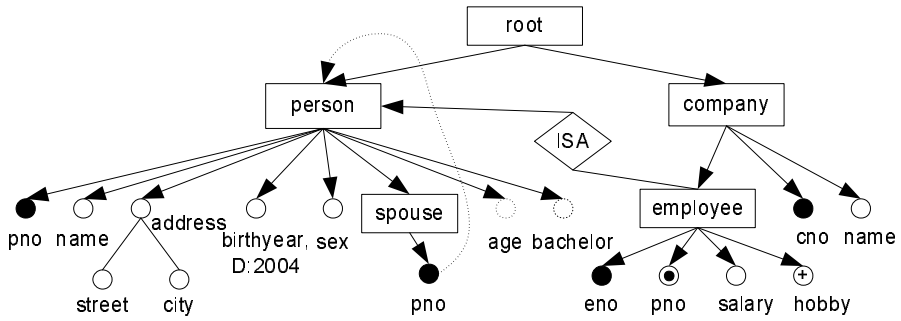

Figure 3.1: Person_Company_Employee ORASS schema diagram

3.1 Schema and Rules

The ORASS schema model [17], which captures more schematic information than any other XML schema models, is used and extended to include the deductive and inheritance features as shown in Figure 3.1. Notice in ORASS schema model, there are object classes and attributes which are different from elements and attributes as in XML. The elements with ID attribute in XML are mapped to object classes while the remaining elements in XML are mapped to attributes in ORASS. All the attributes in XML are mapped to attributes in ORASS. In the schema diagram, *root* is the XML document root, which contains object class *person* and *company* shown in rectangles.

The *person* object class has an attribute *pno* which uniquely identifies the person object and the *pno* identifier attribute is shown as a filled circle in the schema diagram. It also includes attribute *name*, composite attribute *address* which contains *street* and *city*, *birthyear* with a default value 2004 indicated by prefixing character D before 2004, *sex*, and object class *spouse* which contains *pno* referring to person.

Since birthyear is already defined in the person class, to avoid duplicate

information, it is not advisable to define the age for person. Another reason is that the age attribute needs to be updated once a year. However, the age attribute is often used as part of a query. Therefore, it is better to define age as a derived attribute. The *person* object class has two derived attributes, *age* and *bachelor* which are indicated by dashed circles in the schema diagram. Derived attribute *age* is used to calculate the age of a person from the birthyear. Derived attribute *bachelor* is used to indicate whether a person is a bachelor or not.

The *company* object class has an attribute *cno* which uniquely identify the company object. It also includes attribute *name* and object class *employee*.

The *employee* object class who is a subclass of *person* object class inherits all the attributes, object classes, and derived attributes from *person* class. The inheritance relationship is denoted by *ISA* diamond in the schema diagram. The employee object class has its own identifier *eno*, the candidate identifier *pno* indicated by a filled circle inside a circle referring to the person object, and two extra attributes *salary* and *hobby*. *Hobby* is a multi-valued attribute as indicated by "+" which means an employee may have one or more hobbies.

In this example, we can see the two new features that are not present in XML databases: *derived attribute* and *class inheritance*. Class inheritance is supported in current XML Schema [20]. We now highlight how to define the derived attributes of object classes. In object-oriented programming languages, methods are defined using functions or procedures and are encapsulated in class definitions. In deductive databases, rules are used instead of functions and procedures. By analogy, derived attributes or methods in

XDO2 are defined using deductive rules and encapsulated in class definitions. In the following, we use deductive rules to define the method *age* and *bachelor* encapsulated in object class *person*.

$\$p/age : \$a$:- $/root/person : \$p/birthyear : \$b, \$a = 2004 - \$b$.

This rule says if there is a *person* element under the *root* element, and the *person* has sub-element *birthyear*, then the age is equal to 2004 minus the birthyear. In the method *age* above, the notation ":-" means if a substitution of all variables to values makes the right hand side true, then the left hand side is also true. The notation ":" binds the value of the left hand side to the right hand side. If the left hand side is an object class, then the right hand side binds to the object identifier, such as $p binds to the person's identifier. Otherwise, it binds to the value of the left hand side. The single-valued variable is denoted by a "$" followed by a string literal.

$\$p/bachelor : true$:- $/root/person : \$p/[sex : \text{“}Male\text{”}, not(spouse : \$s)]$.

This rule says if a *person* element under *root* element has an attribute *sex* with string value *"Male"*, and this same person does not have *spouse*, then the derived attribute *bachelor* of the object class *person* has boolean value *true*. The two boolean value *true* and *false* are reserved in the language. The notation "[ ]" in the *bachelor* method above is used to group the attributes, elements or methods which are directly defined under the same parent element, such as *person* in this case. The notation "not" negates the existence of the enclosed expression and is similar to not-predicate.

```
<root>
   <person pno="p1">
      <name>John</name>
      <address>
         <street>King</street>
         <city>Ottawa</city>
      </address>
      <birthyear>1975</birthyear>
      <sex>Male</sex>
   </person>
   <person pno="p2">
      <spouse pno="p3" />
      <name>Mike</name>
      <address>
         <street>Albert</street>
         <city>Ottawa</city>
      </address>
      <birthyear>1954</birthyear>
      <sex>Male</sex>
   </person>
   <person pno="p3">
      <spouse pno="p2" />
      <name>Mary</name>
      <address>
         <street>Albert</street>
         <city>Ottawa</city>
      </address>
      <birthyear>1958</birthyear>
      <sex>Female</sex>
   </person>
   <company cno="c1">
      <name>Star</name>
      <employee eno="e1" pno="p1">
         <salary>6000</salary>
         <hobby>Tennis</hobby>
         <hobby>Soccer</hobby>
      </employee>
      <employee eno="e2" pno="p2">
         <salary>4000</salary>
         <hobby>Tennis</hobby>
      </employee>
   </company>
</root>
```

(a) XML extensional database

```
% Rule R1 defines that the age of a
% person is 2004 minus his/her
% birthyear.

(R1)  $p/age : $a :- /root/person : $p/
         birthyear : $b, $a = 2004 - $b.

% Rule R2 defines that a person is a
% bachelor if he is a male and without
% spouse.

(R2)  $p/bachelor : true :- /root/
         person : $p/[sex : "Male",
         not(spouse : $s)].
```

(b) XML intentional database

```
employee ISA person
by employee.pno ISA person.pno
```

(c) XML class hierarchy relationships

Figure 3.2: Person_Company_Employee database

3.2 Data and Query

The data or instance of the *Person_Company_Employee* database is shown in Figure 3.2. For highlighting, we also include the definitions of deductive rules and definitions of class hierarchy relationships which are defined in the schema of the XDO2 database. There are three parts to the database: the *XML extensional database*, the *XML intentional database*, and the *XML class hierarchy relationships*. The XML extensional database contains the XML data element facts with their tree structure. The XML intentional database contains the deductive rules which can be used to derive new XML data elements or attributes from the extensional database. The XML class hierarchy relationships define the object class hierarchy in the database such as *employee* is a subclass of *person*. Storing the deductive rules and class hierarchy relationships in the XML database system, enables querying using deductive rules and the class hierarchy, as shown in the following example.

Example 3.1. This query retrieves the employees' age and salary who are a bachelor, with *age* less than 30, and *salary* larger than 5000.

$$/db/youngRichBachelor : \$e/[age : \$a, payroll : \$s] \Leftarrow /root/company/$$
$$employee : \$e/[age : \$a, bachelor : true, salary : \$s], \$a < 30,$$
$$\$s > 5000.$$

Notice the query format is similar to the deductive rule used to describe methods. The notation "$\Leftarrow$" separates the return format of the query from the query and conditional part. The left hand side is used to define the XML result format, like in the *return* clause in XQuery, and the right hand side is the query and the conditional parts like the *for*, *let* and *where* clauses in XQuery. Therefore, our XDO2 query language is more simple and compact

with only one line of some predicate expressions instead of FLWR clause in XQuery. With the deductive rules and the inheritance feature defined in the XML database, the user can directly query the attributes or methods both in *employee* and its superclass *person*, such as *age* and *bachelor* in the example.

Using the XDO2 database in Figure 3.2, only employee *'e1'*, whose *pno* is *'p1'* satisfies the conditions. Using the *youngRichBachelor* element and its two sub-elements *age* and *payroll* for the result format, the query result is as follows.

```
<db>
   <youngRichBachelor eno="e1">
      <age>29</age>
      <payroll>6000</payroll>
   </youngRichBachelor>
</db>
```

Notice $e binds to the object identifier value of the employee object, i.e., eno value. The attributes of *youngRichBachelor*, *age* and *payroll* are from the derived attribute *age* of *person* object *'p1'*, and *salary* of *employee* object *'e1'* respectively.

Chapter 4

XDO2 Language Features

In the previous chapter, we have already given a typical XDO2 database example to motivate the discussion of our XDO2 language. In that example, we present some important features such as the *deductive rules* and *inheritance* to simplify the query. However, the language features are not covered completely. In this chapter, we provide a more detailed coverage of the features that XDO2 language has.

As we know, the XDO2 language combines the techniques from XTree [10], deductive databases and object-oriented databases. Therefore, most of the salient features of XDO2 language are from the three paradigms. From XTree, we can get a more compact and simple query language than XPath [13] since XTree is designed to have a tree structure while XPath does not. XTree can also be used to define the query return format which effectively avoids the nested structure as in XQuery [6]. In XTree, multi-valued variables are explicitly indicated, and their values are uniquely determined while in XQuery, there is no syntactic difference between single-valued variables and multi-valued variables. These multi-valued variables are manipulated to have

some natural built-in functions in an object-oriented fashion. XTree can also explicitly distinguish the XML structure or schema from the XML data value, supporting schema querying naturally. Unlike in XQuery, the schema of XML and XML data are bound to the variables together.

In deductive databases, negation is one of the most important features which negates the predicate. It makes the language more meaningful and powerful. In XDO2 language, we will use the not-predicate as presented in [26] for negation querying instead of using logical negation as in deductive databases and in XQuery. It will be explained in more detail later. Another important feature is that recursive queries are supported naturally using recursive rules which makes the language more powerful.

There are many features in the object-oriented paradigm. Some of the main features include object identity, complex object, typing, class inheritance, multiple inheritance with conflict handling mechanism, overriding, blocking, method encapsulation, method overloading, late binding, and polymorphism. As we know, complex object is the feature used to model real world objects. However, in XDO2, we consider the XML document as a tree structure and use the XML model as the basis. Therefore, we are not going to talk about the complex object feature in XDO2.

This chapter is organized as follows. Section 4.1 presents those XDO2 features from XTree. It includes four features. One is compact and simple properties compared to XPath. Another is single return format expression which differs from nested queries with plain text as in XQuery. The third one is the explicit use of multi-valued variables with its object-oriented fashion built-in functions. And the fourth one is separating structure from value.

Section 4.2 covers the XDO2 deductive features which includes negation and recursion. Section 4.3 covers the XDO2 features from object-oriented databases.

4.1 XDO2 Features from XTree

The major contribution of XTree is the use of square bracket [ ] to group the same level attributes and/or elements together so that multiple navigational paths are grouped together. As a result, the XTree [10] language corresponds to the XML tree structure and is more suitable for XML querying than XPath [13] which is designed as one navigational path. Since our XDO2 language is designed based on XTree, our XDO2 query language support the features from XTree technique naturally. Due to the XDO2 language has features from deductive and object-oriented paradigm, there are some notation differences and extensions from XTree (XTreeQuery) which are summarized as follows,

1. *Stru : value* term is used in XDO2 instead of *stru* $\rightarrow$ *value* and *stru* $\leftarrow$ *value* used in querying and result format as in XTreeQuery.

2. There are *query*, *where* and *construct* clauses used in XTreeQuery while there is only one clause with format *result* $\Leftarrow$ *conditions* in XDO2. *result* is similar to the *construct* clause in XTree and the *conditions* consist of a list of expressions which are equivalent to *query* and *where* clauses in XTree.

3. In XML, the order of sub-elements is important and thus we introduce list-valued variables in XDO2 as an extension of set-valued variables in

XTree.

4.1.1 Simple and compact

An XPath [13] expression is just a linear path to a target XML node set. In the querying part of a query, one XPath expression can only bind one variable. However, XDO2 use square bracket [ ] to group the same level attributes, elements and methods together so that it has a tree structure which is similar to the structure of an XML document. In the querying part of a query, one XDO2 expression can bind multiple variables. Therefore, when a user requires data from many paths, only one XDO2 expression is needed, and the XPath expressions will be much more complex.

Example 4.1. To find the year and title of each book, and its authors' last name and first name.

XDO2 expression:

$/bib/book/[@year : \$y, title : \$t, author/[last : \$l, first : \$f]]$

XPath expressions:

$book in /bib/book, $y in $book/@year, $t in $book/title, $author in $book/author, $last in $author/last, $first in $author/first

From the two expressions above, we can see the first XDO2 expression is much more simple and compact than the second XPath expressions although they express the same meaning. We also notice the XDO2 expression has only four variables defined which are the user required information while the XPath expressions need six variables. The reason is because the XPath expressions need extra $book and $author defined to keep the correlation between the variables.

4.1.2 Compact query return format

XDO2 expressions can not only be used to bind variables in the querying part, but also can be used to define the result format. For the symbol ":", if the right side is a single-valued variable, we just bind the value in the current iteration to the left side; if the right side is a multi-valued variable, we bind all the values in the set or list to the left side. Unlike the return clause in XQuery that often mixes XML plain text, enclosed expression and even sub-queries, here the result construction part is just an XDO2 expression without nesting, which is very simple and easy to read.

Example 4.2. To list the titles and publishers of books which are published after 2000.

XDO2 query expression:

/result/recentbook/[title : $t, publisher : $p] ⇐ /bib/book/[@year : $y,
title : $t, publisher : $p], $y > 2000.

XQuery expression:

for $book in /bib/book, $y in $book/@year, $t in $book/title,
$p in $book/publisher
where $y > 2000
return <result><recentbook>{$t}{$p}</recentbook></result>

As we can see from the result construction part of both XDO2 and XQuery, only one XDO2 expression is used for the query result format while in XQuery, the XML plain text (element tags) is mixed with the XPath expressions.

4.1.3 Aggregate functions

In XDO2, multi-valued variables are explicitly indicated and include list-valued and set-valued variables. List-valued variables are explicitly indicated by angle bracket < >. Set-valued variables are explicitly indicated by curly braces { }. Object-oriented fashion built-in aggregate functions are supported through the use of multi-valued variables. Suppose a multi-valued variable {$num} or <$num> binds to a set or a list of numbers, then the aggregate functions supported are as follows,

{$num}.max() or <$num>.max() maximum value in the set or list

{$num}.min() or <$num>.min() minimum value in the set or list

{$num}.count() or <$num>.count() number of items in the set or list

{$num}.sum() or <$num>.sum() sum of values in the set or list

{$num}.avg() or <$num>.avg() average value of items in the set or list

Example 4.3. List the title of the books which has more than 1 author.

XDO2 query expression:

/result/multiAuthorBook/title : $t ⇐ /bib/book/[title : $t, author : <$a>],

<$a>.count() > 1.

XQuery expression:

for $book in /bib/book, $t in $book/title

let $a in $book/author

where count($a) > 1

return <result><multiAuthorBook>{$t}</multiAuthorBook></result>

In XQuery, there is no syntactic difference between single-valued variables and multi-valued variables. The multi-valued variables are defined in the *let* clause. However, in XDO2, the multi-valued variables are explicitly indicated

by surrounding angle brackets < > (curly braces { }). We also noticed that the object-oriented fashion built-in aggregate functions are supported in XDO2, such as *{$a}.count()* in this example. However, in XQuery, the built-in aggregate functions are supported in function based fashion, such as *count($a)*.

4.1.4 Separating structure and value

One of the disadvantages of XPath is that the variable which binds to one XPath expression denote both the structure of the element (attribute) and the value of the element (attribute). While in the XDO2 language, we can separate the two parts using term *stru : value*. On the left of : symbol *stru* denotes the structure while the right of : symbol *value* denotes the value.

Example 4.4. To obtain some sub-element with value "John" in some person element.

XDO2 query expression:

⇐ /root/person/$ele : "John"

XQuery expression:

*for $b in /root/person/**

where string($b) = "John"

return local-name($b)

Notice we omit the result format expression in the XDO2 query and the variable $ele with its value pairs are returned. As we can see from this example, in XDO2, the XML element tags (attribute names) are separated from the values naturally because of the term structure *stru : value* used. However, in XQuery, we have to use symbol "*" (all) since the structure is

unknown and have to use two built-in functions *string()* and *local-name()* to get the values and element tags (attribute names) respectively.

4.2 XDO2 Features from Deductive Databases

In deductive database, deductive rules are used to derive new information. There are two important issues in deductive database. One is negation which negates the predicate and makes the query more meaningful or powerful. We will use the not-predicate [26] for querying instead of conventional logical negation. The other one is recursive querying which directly uses the recursive deductive rules and makes the query more powerful. In this section, we present the two issues in our XDO2 languages and make these queries possible.

4.2.1 Negation

In deductive databases, negation makes the rules more powerful and queries more meaningful. However, it complicates the query's interpretation and evaluation. To represent negation in XDO2, we choose the not-predicate [26] instead of the conventional logical negation symbol "$\sim$", which just negates the boolean expression. It has been noted in [26] that the not-predicate is not always equivalent to "$\sim$" in negation expression. The main difference between the not-predicate and "$\sim$" lies in the interpretation of the uninstantiable variables (i.e. variables that do not appear in any positive expression in the body of the rule or query) in the negation expression. Otherwise, they are equivalent. Using the not-predicate, the uninstantiable variables

are existentially quantified while they are universally quantified using "$\sim$". We illustrate the difference in example 4.5 as follows.

Example 4.5. Consider a query that retrieves all the bachelors, i.e., a male person without a spouse.

(Q4.5.1) $/db/bachelor : \$p \Leftarrow /root/person : \$p/sex : \text{“}Male\text{”},$
$\$p/not(spouse : \$s).$

Using the not-predicate, the uninstantiable variable $s is existential in nature and Q4.5.1 is interpreted as follows,

$\forall \$p(/root/person : \$p/sex : \text{“}Male\text{”} \wedge \sim \exists \$s(\$p/spouse : \$s)$
$\rightarrow /db/bachelor : \$p).$

This interpretation says for any person $p, if $p is a male and there does not exist a spouse of $p, then $p is a bachelor. This interpretation corresponds to the user's meaning.

However, in order to express the query using "$\sim$" for negation expression, it is not correct if we simply change "not" in Q4.5.1 to "$\sim$" as shown in Q4.5.2,

(Q4.5.2) $/db/bachelor : \$p \Leftarrow /root/person : \$p/sex : \text{“}Male\text{”},$
$\$p/ \sim (spouse : \$s).$

Using "$\sim$", the uninstantiable variable $s is universally quantified and Q4.5.2 is interpreted as follows,

$\forall \$p \forall \$s(/root/person : \$p/sex : \text{“}Male\text{”} \wedge \sim (\$p/spouse : \$s)$
$\rightarrow /db/bachelor : \$p).$

This interpretation says for any person $p, for any $s, if $p is a male and $p do not have spouse relationship to $s, then $p is a bachelor. So only those person who have spouse relationship to everything do not belong to

the result and the others belong to the result. Therefore, this interpretation does not correspond to the user's meaning. On the other hand, from Clark's safe computation rule [14], Q4.5.2 is defined as a not safe query since there exist uninstantiable variables in the negation expression.

In order to make the query safe while still use "$\sim$" as negation expression, a new deductive rule is needed to express the query as follows,

(R3) $\$p/married : true$:- $/root/person : \$p/spouse : \$s$.

(Q4.5.3) $/db/bachelor : \$p \Leftarrow /root/person : \$p/sex :$ "$Male$",

$\$p/ \sim (married : true)$.

With the help of a deductive rule R3 that hides the uninstantiable variable $s from the query, Q4.5.3 is a safe query and the interpretation is what the user required. However, using the "$\sim$" as negation expression, users may need to define new rules for some simple queries. This is not acceptable as users should not need to define and add deductive rules during querying. Moreover in a multi-user environment, this could lead to unpredictable results if different users declare the same rule more than once differently. Therefore, we use the not-predicate instead of the conventional logical negation symbol "$\sim$" in a negation expression.

As we know, XQuery [6] provides a function *not()* which needs a boolean value as its argument and similar to the meaning of "$\sim$", and it does not support the not-predicate operator. The function *not()* is usually combined with *some* and *every* quantifiers for those universal and existential queries. However, by using the logic not-predicate operator alone in XDO2, we can achieve the same expressive power and make our queries more simple and compact. Using not-predicate, nested negation and negation on sub-tree

features are possible and shown in the two examples as follows.

Example 4.6. Consider the following query that retrieves the company name of companies where each employee of the company has hobby "Tennis".

XDO2 query expression:

$/db/allLikeTennisCom : \$n \Leftarrow /root/company : \$c/name : \$n,$

$\$c/not(employee/not(hobby : \text{“}Tennis\text{”}))$.

XQuery expression:

for $c in /root/company

where EVERY $e IN $c/employee SATISFIES

SOME $h IN $e/hobby SATISFIES string($h)=“Tennis”

return <db><allLikeTennisCom>{string($c/name)}

</allLikeTennisCom></db>

Notice in the XDO2 query, the company may have many employees and employees may have many hobbies. The interpretation of this query says if the company does not exist an employee who does not have hobby of tennis, which is equivalent to say that each employee has hobby of tennis, then the company's name is in the result. Notice in the nested negation term, there is an uninstantiable variable omitted for employee. That is in the negation term *not(employee/not(hobby : “Tennis”))*, *employee* can be equivalently written as *employee : $e* and $e is the uninstantiable variable. Therefore, the not-predicate should be used instead of "∼" for the first *not()* operator. But for the second *not()* operator, which is the one of *not(hobby : “Tennis”)*, since no uninstantiable variable exists in the term, we can use *∼(hobby : “Tennis”)* instead.

The equivalent XQuery expression is also given. As we can see, our XDO2

query using the not-predicate is much more simple and compact compared with the XQuery which needs the key word "EVERY", "IN", "SATISFIES" to express the same meaning.

Example 4.7. Consider the following query that retrieves the companies which do not have employees who have sex "Male" and birthyear 1975.

XDO2 query expression:

$/db/company : \$c \Leftarrow /root/company : \$c/not(employee/$
$[sex : "Male", birthyear : 1975]).$

XQuery expression:

for $c in /root/company
where NOT (SOME $e IN $c/employee SATISFIES
($e/sex = "Male" AND $e/birthyear = 1975))
return <db>{$c}</db>

In XDO2 query, note the term enclosed by the *not()* operator is a sub-tree structure instead of a path. The meaning is that the company does not have the sub-tree pattern of the employee element with two sub-elements: sex with value "Male" and birthyear with value 1975.

The equivalent XQuery is presented in a complicated format. Notice the "NOT" in XQuery is used to negate the boolean expression and needs to combine with SOME and EVERY for those existential and universal queries. Strictly speaking, *$e/sex* which binds to the structure and value together should not compare with the value "Male". But XQuery allows it for simplicity.

4.2.2 Recursion querying

Recursion is a very important feature of deductive databases. In deductive databases, it is natural to have the recursive query using recursive deductive rules. Similarly, in XDO2, we also support recursive deductive rules and make the recursive query possible to extend the expressive power of our XDO2 language.

Example 4.8. Suppose there is a sub-element child directly under the person element and the following deductive rules are defined in the database.

(R4) $\$p/descendant : \$c$:- $/root/person : \$p/child : \$c$.

(R5) $\$p/descendant : \$d$:- $/root/person : \$p/child : \$c$, $\$c/descendant : \$d$.

Notice the variables in the rules are bind to the object identifiers if the left part of the symbol ":" is an object class, such as $c, $p, $d are all bind to the object identifiers in the example. The rule R4 says for each person identified by $p, if $c is his/her child, then $c is a descendant of $p. The rule R5 says if $c is a child of $p, and $d is a descendant of $c, then $d is also a descendant of $p. Note the predicate descendant is recursively defined. Using the rules defined, we can write a recursive query to retrieve all the descendant of a person 'p1' as follows,

$\Leftarrow /root/person : `p1'/descendant : \$d$.

Suppose there is a data instance in Figure 4.1. After we get the fixpoint of the set of rules which computes all the descendants of all the persons using the rules, we can naturally get the result $d = 'p2', $d = 'p3'.

```
<root>
   <person pno="p1">
      <child pno="p2"/>
   </person>
   <person pno="p2">
      <child pno="p3"/>
   </person>
   <person pno="p3"/>
</root>
```

Figure 4.1: An XML instance

4.3 XDO2 Features from Object-Oriented Databases

Object-oriented databases appeared in the late 1980's and the object-oriented technique has been a very hot topic in the past decade. In this section, we will present the object-oriented features used in our XDO2 system.

4.3.1 Object identity

Each object in an object-oriented database has an identifier to uniquely identify an object. In XDO2, we use an object-identifier to identify an object. Therefore, we assume for each object class definition in the schema of XML, an ID required attribute must be defined as the identifier. As in Figure 3.1, the pno is defined as an ID attribute of person object class. The cno is defined as an ID attribute of company object class and the eno is defined as an ID attribute of employee object class.

4.3.2 Typing

In the object-oriented paradigm, strong typing is important and we support typing in XDO2. In the previous examples, we use : symbol to specify the value of the attribute or element. However, a user can use :: symbol to specify the type of the attribute or element by writing the type or single-valued variable on the right side of :: symbol.

Example 4.9. Find the element tag with type string and element tag with type person.

⇐ */person/[$ele1 :: string, $ele2 :: person]*

In this query, we omit the result format expression. Only the variable with its value pairs are returned. We bind the element tag to $ele1 which has type string. We also bind the element tag to $ele2 which has type person. From the database schema diagram of Figure 3.1, $ele1 = name, $ele2 = spouse and $ele1 = sex, $ele2 = spouse satisfy the query.

4.3.3 Inheritance

Inheritance is one of the most important features in object-oriented databases. It enables the reusability of the schema. In XDO2, the inheritance semantics is specified in the XML schema using ISA diamond as in Figure 3.1. The attributes, sub-elements, and encapsulated methods in the superclasses can be inherited by the subclasses. As in Figure 3.1, the employee object class inherits all the properties from its superclass person.

Example 4.10. List the name and age of employee 'e1'.

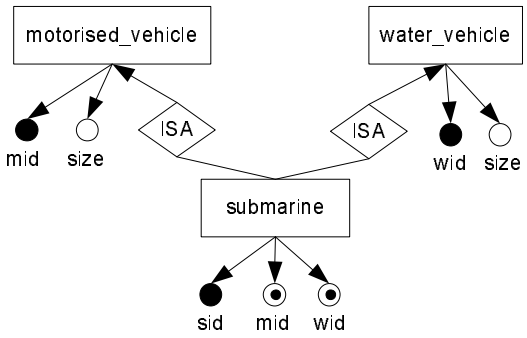

Figure 4.2: Multiple inheritance in ORASS

/db/person_e1/[name : $n, age : $a] $\Leftarrow$ */root/company/employee : 'e1'/ [name : $n, age : $a].*

Notice the sub-element name and method age are not defined directly under the object class employee. The employee object class inherits these properties from its superclass object person and use them directly as normal sub-elements.

4.3.4 Multiple inheritance

Multiple inheritance means one object class can have more than one superclass. A problem that arises when multiple inheritance is supported is the inheritance conflict problem, that is ambiguity may arise when the same property is defined in more than one superclass. XDO2 resolves such conflicts using the explicit selection technique adopted from [28], which has been roughly explained in chapter 2. The explicit selection technique involves indicating explicitly which class a property is to be inherited from.

Suppose Figure 4.2 is part of the ORASS schema model. The class *submarine* is not only a subclass of *motorised_vehicle* but also a subclass of

water_vehicle. Since *size* is defined in both the class *motorised_vehicle* and class *water_vehicle*, there is a conflict and the *submarine* class does not know which *size* attribute it should inherit. So in the schema, we need to explicitly select which *size* attribute to inherit. This can be achieved by the following statements,

submarine ISA motorised_vehicle, water_vehicle
by submarine.mid ISA motorised_vehicle.mid
by submarine.wid ISA water_vehicle.wid
with size INHERITED water_vehicle

The above statements define that the *submarine* is a subclass of *motorised_vehicle* and *water_vehicle* and the attribute *size* is inherited from class *water_vehicle* instead of other classes.

If there is a conflict and there is no conflict resolution declaration, then by default the property is inherited from the first superclass in the superclass list. In this case without the statement of *with size INHERITED water_vehicle*, the conflicting attribute *size* is inherited from class *motorised_vehicle* by default.

Other techniques to resolve multiple inheritance conflicts such as redesigning the schema, removing redundant ISA relationship, redefining an overload property, renaming properties, factoring attributes to a general class are roughly explained in chapter 2 and can be found in [28].

4.3.5 Overriding

When a subclass defines some attributes which have the same name as the attributes in its superclass, the attributes defined in the subclass override the

attributes defined in the superclass and it is known as *attribute overriding*. *Method overriding* is defined similarly as long as the two methods have the same signature (same method name, same number of arguments and same type of the arguments in order). We can support the overriding feature in XDO2 similarly. Because the methods in XDO2 can be considered the same as the derived attributes and the method overriding is the same as attribute overriding.

4.3.6 Blocking

When a subclass inherits the properties from its superclass, by default all the properties of the superclass, including the attributes and methods are inherited. But in reality, it is possible that the subclass does not want some property to be inherited. Therefore, the blocking technique is used here to handle this problem and it can block the properties to be inherited from the superclass. As the technique used in [32], we can redefine the property with a return class of *none*. Then this property is blocked in the current class as if never defined.

Refer to the example as in Figure 3.2, if the class *employee* want to block the *address* property from the class *person*, the following statements can be used and defined in the schema.

employee ISA person

by employee.pno ISA person.pno

with address BLOCKED FROM person

The above statements define that the *employee* is a subclass of *person* and the attribute address is blocked from person as if the address attribute

is never defined in employee class. Notice we explicitly specify which class *address* is blocked from since multiple inheritance is possible. Therefore, we can also use blocking technique to block the conflicting properties from some classes to resolve the multiple inheritance conflicting problem.

4.3.7 Method encapsulation, overloading, and late binding

Without the object-oriented methods encapsulated or supported in the XML database, users have to write their own method in the query system to query the XML data. This will make the query more complicated and prone to errors. By using the deductive rule methods in the XDO2 database system, the XDO2 query language can be simplified by using the method directly like normal properties.

Example 4.11. List all the people who are older than 40 years old and a bachelor.

$$/db/oldBachelor : \$p \Leftarrow /root/person : \$p/[age : \$a,\ bachelor : true],\ \$a > 40.$$

Refer to the Figure 3.2, in this query, the methods age and bachelor which are defined under class person using deductive rules are directly used in the query language. Users do not need to define the methods in the query system but can take the derived properties as a normal sub-element in the query.

In object-oriented paradigm, *method overloading* means methods may have the same method name with different arguments. So that one method name can have multiple different interpretations depends on the arguments

list. In the XDO2 language, we support method overloading naturally since we use deductive rules to define the method. The head of the deductive rule is the object class, the method name, and the return variables or values. The pass in arguments are in the body part of the deductive rule. So two methods with same name are considered as one method with different interpretations, such as the method *descendant* in example 4.8.

Because of inheritance, a subclass may implement a method to *override* that in its superclass. Method resolution that determines which implementation is associated with a given method name and class at runtime, is known as *late binding*. Suppose manager ISA employee and *bonus()* is a method implemented in both manager and employee. When a *bonus()* message is sent to an instance of employee who is also a manager, the *bonus()* method in manager class should be executed instead of the *bonus()* method in employee class. In our XDO2 system, we support late binding naturally since we do not define the type of the variables that are used in the deductive rules or query. The XDO2 query evaluation combines the compile step and runtime step so that the method resolution is determined at runtime.

4.3.8 Polymorphism

As we have stated, polymorphism is equivalent to method late binding in programming language definitions and polymorphism has some other meanings which means multiple forms. It makes objects with common characteristics organized together. A good application of polymorphism is in the XML Schema definition. For example, given an XML document with element root and a list of person elements. The XML Schema for element root may be as

follows.

<xs:element name = “root” type = “rootType” />

<xs:complexType name=“rootType”>

<xs:element name=“person” type=“personType” minOccurs=“0”

maxOccurs=“unbounded” />

</xs:complexType>

From the schema definition above, a list of person elements is expected under the root element. However, if there are some subclasses of person, such as employee and student defined, then we can have employee and student elements as the sub-elements of root if polymorphism is supported in the system. It is also considered as a valid document to the XML Schema.

Since typing is supported in XDO2 system, it is easy to support the polymorphism feature. When a type A data is queried, then data of type A and any data of types that are subtypes of A are considered as valid data.

Chapter 5

XDO2 Language Syntax

In the previous chapters, we have already shown many XDO2 deductive rules and queries. In this chapter, we will define the syntax of the XDO2 language formally. For simplicity, we will not consider the typing feature and "//", which means multiple levels down.

The *values* are defined so that they correspond to the XML document text data, such as the string data of element sex, integer data of element birthyear, set value data of attribute children of type IDREFS, list value data of element hobby of a person. Particularly, NULL can be used for the empty value.

The *terms* are recursively defined so that they can be used to form the expressions. *Attribute* term, *element* term, *attribute_value* term, and *element_value* term are defined so that they correspond to the XML document attribute or element with optional value bound. *Negation* term is recursively defined so that XDO2 language supports negation querying using not-predicate [26]. *Grouping* term is recursively defined so that XDO2 language supports tree structure querying as in XTree [10]. Finally, *path* term is re-

cursively defined similar to navigational path as in XPath [13].

Using the terms, *expressions* are defined. The *deductive rule* and *query* are just composed of a list of expressions.

Let $\mathbb{U}$ be a set of URLs, $\mathbb{C}$ be a set of constants, and $\mathbb{V}$ be a set of variables.

The set of constants $\mathbb{C}$ contain strings enclosed by " ", integers, real numbers, two boolean values and object identifiers enclosed by ' '. The object identifiers are all the values of identifiers with ID type in the XML data. Unlike in XML data both the value of ID type and string type of XML attribute are enclosed by " ", in XDO2 the object id which denotes an object is enclosed by ' ' and thus is different from the string data enclosed by " ".

The set of variables $\mathbb{V}$ are partitioned into single-valued and multi-valued variables. Single-valued variables have format $S where S is a string literal. Multi-valued variables include set-valued variables with format {$S} and list-valued variables with format <$S> where S is a string literal. Set-valued variables denote a set of items without duplicates and the order does not matter. In XML, only attributes with IDREFS type can have a set of values and the set-valued variables are used here only. List-valued variables denote a list of items with duplicates possible, where order does matter. In XML, the list-valued variables are used to denote the values of the multi-valued elements such as hobbies of an employee. Particularly, $_ is defined as the anonymous single-valued variable, {$_} is defined as the anonymous set-valued variable and <$_> is defined as the anonymous list-valued variable.

Definition 5.1. The *values* are defined as follows,

1. null is a *null* value.

2. Let $c \in \mathbb{C}$ be a constant. Then c is a *constant* value.

3. A set of object identifiers is a *set* value. Set values are only used to denote the values of XML attributes which have type IDREFS. A set value is only possible to have a set of object identifiers.

4. A list of constant values is a *list* value. List values denote the value of the multi-valued XML element, such as the value of hobbies.

Example 5.1.

Constant values: "John", 30, 4.8, true, false, 'p2'

Set values: {'e1'}, {'p1', 'p2', 'p3'}

List values: <"John", "Mary">, <68742779, 68742556>

Definition 5.2. The *terms* are defined recursively as follows,

1. Let t be an XML attribute name. Then @t is an *attribute* term.

2. Let t be an XML element tag. Then t is an *element* term.

3. Let X be an attribute name or a single-valued variable, and Y a constant value, a set value, a single-valued variable or a set-valued variable. Then @X : Y is an *attribute_value* term, and Y denotes the value of the attribute X.

4. Let X be an element tag or a single-valued variable, and Y a constant value, a list value, a single-valued variable or a list-valued variable. Then X : Y is an *element_value* term, and Y denotes the value of the element X.

5. Let X be a term. Then not(X) is a *negation* term.

6. Let $X_1, \ldots, X_n$, $(n \geq 2)$ be a set of terms. Then $[X_1, \ldots, X_n]$ is a *grouping* term.

7. Let $X_1, \ldots, X_n$, $(n \geq 2)$ be a set of terms and $X_1, \ldots, X_{n-1}$ are not grouping terms or negation terms. Then $X_1/\ldots/X_n$ is a *path* term.

Example 5.2.

Attribute terms: @pno, @birthyear

Element terms: sex, address, age

Attribute_value terms: @birthyear : $y, @$attr : {'p2', 'p3'}, @$x : {$y}

Element_value terms: name : $n, author : <$a>, $ele : "Male"

Negation terms: not(spouse : $s), not(employee/not(hobby : "Tennis"))

Grouping terms: [age : $a, bachelor : true, salary : $s],
[spouse : $s, name : $n, address/street : $st]

Path terms: person/name : $n, root/[company : $c, person : $p]

Definition 5.3. The *expressions* are defined exclusively as follows,

1. Let $u \in \mathbb{U}$ be a URL and P be a path term. Then (u)/P is an *absolute path* expression. If URL u is the default one, such as standard input, we can omit it and use /P instead.

2. Let X be a variable or an object id, and P be a term. Then X/P is a *relative path* expression. An *instantiable relative path* expression is a relative path expression X/P where either X is some object id, or the variable X has been defined in some positive terms (which is not a negation terms or inside a negation term).

3. *Arithmetic, logical* expressions are defined using variables, values, aggregate functions and operators in the usual way. *Instantiable arithmetic, logical* expressions are arithmetic, logical expressions such that all the variables inside are defined in some positive terms.

Example 5.3.

Absolute path expression:

(http://www.abc.com/root.xml)/root/person : $p

Relative path expressions: $u/salary, $p/name, 'p1'/age : $a

Arithmetic or logical expressions: $a = $b * 2, $age > 30,

<$s>.distinct().count() = 3

Definition 5.4. A deductive *rule* has the form

H :- $L_1, \ldots, L_n$.

where H is the head and $L_1, \ldots, L_n$ is the body of the rule. H is a positive instantiable relative path expression and L_1, ..., L_n are either absolute path expressions or instantiable expressions.

Example 5.4.

$p/age : $a :- /root/person : $p/birthyear : $b, $a = 2004 - $b.

$p/bachelor : true :- /root/person : $p/[sex : "Male", not(spouse : $s)].

Definition 5.5. A *query* has the form

$R \Leftarrow L_1, \ldots, L_n$.

where R is the result format expression and $L_1, \ldots, L_n$ are the query or conditional expressions. R is a positive absolute path expression and $L_1, \ldots, L_n$ are either absolute path expressions or instantiable expressions. If there is no result format expression specified, we use

$$\Leftarrow L_1, \ldots, L_n.$$

instead.

Example 5.5.

/db/youngRichBachelor : $e/[age : $a, payroll : $s] $\Leftarrow$ /root/company/
employee : $e/[age : $a, bachelor : true, salary : $s], $a < 30,
$s > 5000.

Chapter 6

XDO2 Language Semantics

In this chapter, we will formally define the semantics of the XDO2 language.

Importantly, given a query, we need to define the semantics of the query precisely. Our main idea is: firstly, we get all the deductive rules that are used by the query; secondly, we need to find a minimal model or a least fixpoint (an XML database) for these deductive rules; thirdly, we use the minimal model to retrieve and format the results. The challenging part is the second step for finding a minimal model for a set of deductive rules. In this step, we need to define a model first and then to define the minimal model. There are two important definitions needed to be introduced, one is *part-of* definition used for XML tree structure and the other one is the *satisfaction* definition used for whether an XML tree data satisfies the deductive rule or not.

Definition 6.1. A list $L \equiv< L_1, \ldots, L_m >$ is a *sublist* of another list $L^{'} \equiv< L^{'}_1, \ldots, L^{'}_n >$ if and only if $m \leq n$ and $\exists$ a list $< L^{'}_{a_1}, L^{'}_{a_2}, \ldots, L^{'}_{a_m} >$, such that $1 \leq a_1 < a_2 < \ldots < a_m \leq n$, and $L_i = L^{'}_{a_i}$ for $1 \leq i \leq m$.

A list L_1 is a *sublist* of another list L_2 means L_1 is contained in L_2

pertaining the order.

Definition 6.2. A *ground substitution* θ is a mapping from the set of variables $\mathbb{V}$ - {$_ , {$_}, <$_>} to the set of all attribute names, element tags and the set of all the values except null. Single-valued variables are mapped to attribute names, element tags and constant value. Set-valued variables are mapped to set value and list-valued variables are mapped to list value.

Notice we disallow anonymous variables when we deal with semantics. If anonymous variables are used, we should change them to named variables where the names have never been used by other variables.

Definition 6.3. A *ground term* is a term with a ground substitution θ such that each variable is instantialized to an attribute name, element tag, or a value. *Ground expression* is defined similarly.

In the definition of terms in previous chapter, there may have variables in the terms. Ground term is defined here so that each variable is initialized and no variables are inside the ground term.

Definition 6.4. A value v is *part-of* another value $v^{'}$, denoted by $v \preceq v^{'}$, if and only if one of the following holds,

1. v is null.

2. v is a constant value.

   - $v^{'}$ is a constant value and $v^{'} = v$;

   - $v^{'}$ is a list value $v^{'} \equiv < v_1^{'}, \ldots, v_n^{'} >$, and $\exists$ some value $v_j^{'}$ for $1 \leq j \leq n$ such that $v_j^{'} = v$.

3. v is a set value. $v^{'}$ is a set value and $v^{'} = v$.

4. v is a list value. $v^{'}$ is a list value such that v is a sublist of $v^{'}$.

Example 6.1. Some examples on *part-of* of values,

"John" $\preceq$ "John"

68742779 $\preceq$ <68742556, 68742779>

{'p1', 'p2'} $\preceq$ {'p1', 'p2'}

<"John", "Mary"> $\preceq$ <"John", "Smith", "Mary">

The definition of *part-of* is very important in the language semantics definitions. Informally, a tree A is part-of a tree B means we can find a tree pattern A inside B. In this definition, we defined the part-of for values. In the following, we will also define the part-of for ground terms, part-of for ground absolute path expressions, part-of for XML documents, and part-of for XML databases. Notice we define a set value is part-of another set value only when they are equal.

Definition 6.5. A ground term p is *part-of* of another positive ground term $p^{'}$, denoted by $p \preceq p^{'}$ and its negation, denoted by $p \npreceq p^{'}$, if and only if one of the following holds,

1. p is an attribute or attribute_value term with $p \equiv p_1 : v_1$ (v_1 is null if p is an attribute term).

   - $p^{'}$ is an attribute or attribute_value term with $p^{'} \equiv p_1^{'} : v_1^{'}$ such that $p_1^{'} = p_1$ and $v_1^{'} \preceq v_1$;
   - $p^{'}$ is a grouping term with $p^{'} \equiv [p_1^{'}, \ldots, p_n^{'}]$ and $\exists p_i^{'}$ for $1 \leq i \leq n$ such that $p \preceq p_i^{'}$.

2. p is an element or element_value term with $p \equiv p_1 : v_1$ (v_1 is null if p is an element term).

   - $p^{'}$ is an element or element_value term with $p^{'} \equiv p_1^{'} : v_1^{'}$ such that $p_1^{'} = p_1$ and $v_1 \preceq v_1^{'}$;
   - $p^{'}$ is a grouping term with $p^{'} \equiv [p_1^{'}, \ldots, p_n^{'}]$ and $\exists p_i^{'}$ for $1 \leq i \leq n$ such that $p \preceq p_i^{'}$;
   - $p^{'}$ is a path term with $p^{'} \equiv p_1^{'}/\ldots/p_n^{'}$, and $p \preceq p_1^{'}$.

3. p is a negation term with $p \equiv not(x)$ and $x \npreceq p^{'}$.

4. p is a grouping term with $p \equiv [p_1, \ldots, p_m]$. $p^{'}$ is a grouping term with $p^{'} \equiv [p_1^{'}, \ldots, p_n^{'}]$, and $\forall p_i (1 \leq i \leq m)$, $\exists p_j^{'} (1 \leq j \leq n)$ such that $p_i \preceq p_j^{'}$.

5. p is a path term with $p \equiv p_1/\ldots/p_m$.

   - $p^{'}$ is a grouping term with $p^{'} \equiv [p_1^{'}, \ldots, p_n^{'}]$ and $\exists p_i^{'}$ for $1 \leq i \leq n$ such that $p \preceq p_i^{'}$;
   - $p^{'}$ is a path term with $p^{'} \equiv p_1^{'}/\ldots/p_n^{'}$ such that $p_1 \preceq p_1^{'}$ and $(p_2/\ldots/p_m) \preceq (p_2^{'}/\ldots/p_n^{'})$.

Example 6.2. Some examples on *part-of* of ground terms,

@birthyear $\preceq$ @birthyear

@birthyear $\preceq$ @birthyear : 1981

person $\preceq$ person

name $\preceq$ name : "John"

address $\preceq$ address/[street : "King Street", city : "Ottawa"]

person : 'p1' $\preceq$ person : 'p1'/name : "John"

root/person/[name : "John", address, birthyear : 1975] $\preceq$

root/person : ‘p1’/[name : “John”, address/[street : “King Street”,
city : “Ottawa”], sex : “Male”, birthyear : 1975]
root/not(employee/not(hobby : “Tennis”)) $\preceq$
root/[employee/[hobby : “Tennis”, hobby : “Soccer”], employee/
hobby : “Tennis”]

From the word meaning of “part-of”, users may think that “part-of” has a transitivity property. However, because the not-predicate is possible, the transitivity property fails. For example, *employee/not(hobby : “Tennis”)* $\preceq$ *employee*, and *employee* $\preceq$ *employee/hobby : “Tennis”*. However *employee/not(hobby : “Tennis”)* $\not\preceq$ *employee/hobby : “Tennis”*. Therefore, “part-of” property does not have the transitivity property as the name possibly implies.

Definition 6.6. Let $L = (u)/p$ be a ground absolute path expression and $L^{'} = (u^{'})/p^{'}$ be a positive ground absolute path expression. Then L is *part-of* $L^{'}$, denoted by $L \preceq L^{'}$, if and only if $u = u^{'}$ and $p \preceq p^{'}$.

Theorem 1. Let O be an XML document in XML database XDB, then O can be expressed by a positive ground absolute path expression $(u)/p$. Order information is retained by giving a constraint that terms in grouping term are ordered.

Proof. Let O be an XML document with URL u and tree structure of height n.

1. $n = 1$, let r be the root of O. Let $p = r$, then O can be expressed by $(u)/p$.

2. Suppose when $n \leq k$, O can be expressed by $(u)/p$. When $n = k + 1$, let r be the root of O and $c_1, \ldots, c_m$ be the child elements of r with height at most k. By assumption, $c_1, \ldots, c_m$ can be expressed by terms $p_1, \ldots, p_m$, Then O can be expressed by $(u)/r/[p_1, \ldots, p_m]$. Since $r/[p_1, \ldots, p_m]$ is a path term, so O can be expressed by $(u)/p$ with $p = r/[p_1, \ldots, p_m]$.

□

Our XDO2 query expression has a tree structure using the square bracket [ ] to group the same level attributes or elements. When querying, the terms inside the grouping term is logical "AND", and the order does not matter. However, by giving a constraint that terms in grouping term are ordered, we can naturally express the XML tree with one positive ground absolute path expression.

Definition 6.7. Let O and $O^{'}$ be two XML documents. By theorem 1, O can be denoted as $(u)/p$ and $O^{'}$ can be denoted as $(u^{'})/p^{'}$. Then O is *part-of* $O^{'}$, denoted by $O \preceq O^{'}$, if and only if $u = u^{'}$ and $p \preceq p^{'}$.

Definition 6.8. Let XDB and $XDB^{'}$ be two XML databases, which consists of a set of XML documents. Then XDB is *part-of* $XDB^{'}$, denoted by $XDB \preceq XDB^{'}$, if and only if for each $O \in XDB - XDB^{'}$, there exists $O^{'} \in XDB^{'} - XDB$ such that $O \preceq O^{'}$.

Theorem 2. Every instantiable relative path expression can be transformed to its absolute path expression form.

Proof. Let X/P be an instantiable relative path expression. If X is an object identifier, let R be the associated path from the root to the object

identified by X, then the absolute path expression of X/P is R/P. If X is a variable defined in some positive terms, then X has its associated path identified by R and the absolute path expression of X/P is R/P. □

This theorem is straightforward since instantiable relative path must have its starting variable or object id defined (which has a path from the root to the target node), so by replacing the starting variable or object id with its path, it results in its absolute path expression.

Definition 6.9. Let O be an XML document in XDB, the *inheritance completeness* $O*$ is an extension of O that takes the inheritance, overriding and blocking of attributes, elements, and derived properties into consideration.

Definition 6.10. Let XDB be an XML database. The notation of *satisfaction* (denoted by $\models$) and its negation (denoted by $\not\models$) based on XDB are defined as follows.

1. For a ground absolute path expression $(u)/p$, $XDB \models (u)/p$ if and only if there exists $O \in XDB$ with $O* = (u^{'})/p^{'}$ such that $u = u^{'}$ and $p \preceq p^{'}$.

2. For a ground arithmetic, logical expression ψ, $XDB \models \psi$ if and only if ψ is true in the usual sense.

3. For a deductive rule r of the form H :- $L_1, \ldots, L_n$, where by theorem 2, H is an absolute path expression and $L_1, \ldots, L_n$ are either absolute path expressions, or arithmetic or logical expression, $XDB \models r$ if and only if for every ground substitution θ, $XDB \models \theta L_1, \ldots, XDB \models \theta L_n$ implies $XDB \models \theta H$.

Example 6.3. Let XDB be the XML database in Figure 3.2. Then we have

$XDB \vDash$ /root/person/[name : "John", sex : "Male"]

$XDB \vDash$ /root/company/employee : 'e1'/[birthyear : 1975, salary : 6000]

$XDB \vDash$ /root/company : 'c1'/not(employee/not(hobby : "Tennis"))

$XDB \vDash$ <'e1', 'e2'>.count()>1, 29 = 2004 - 1975, 6000 > 5000

The definition *satisfaction* is defined between the XML database and the expressions at first. The satisfaction for a deductive rule is also defined that when the expressions in the body of the rule are satisfied by the XML database, then the head expression should also be satisfied by the XML database.

Definition 6.11. Let R be a set of deductive rules. A *model* M of R is an XML database XDB that satisfies each rule in R. A model M of R is *minimal* if and only if for each model N of R, $M \preceq N$.

Similar to the model definition for a deductive database with a set of rules, we also define a model for the XML database with a set of deductive rules to be an XML database that satisfies each rule. By using the part-of definition for XML database, we can easily define the minimal model.

Definition 6.12. Let p and $p^{'}$ be two positive ground terms. The *tree join* operator denoted by $\uplus$ is defined as follows,

1. p is an attribute term, element term, attribute_value term, or element_value term.

   - $p^{'}$ is a grouping term with $p^{'} \equiv [p_1^{'}, \ldots, p_n^{'}]$, then $p \uplus p^{'} = [p, p_1^{'}, \ldots, p_n^{'}]$;

- Otherwise, $p \uplus p' = [p, p']$.

2. p is a grouping term with $p \equiv [p_1, \ldots, p_m]$.

   - p' is a grouping term with $p' \equiv [p'_1, \ldots, p'_n]$, then $p \uplus p' = [p_1, \ldots, p_m, p'_1, \ldots, p'_n]$;
   - Otherwise, $p \uplus p' = [p_1, \ldots, p_m, p']$.

3. p is a path term with $p \equiv p_1/\ldots/p_m$.

   - p' is a grouping term with $p' \equiv [p'_1, \ldots, p'_n]$, then $p \uplus p' = [p, p'_1, \ldots, p'_n]$;
   - p' is a path term with $p' \equiv p'_1/\ldots/p'_n$.
     . $p_1 = p'_1$, then $p \uplus p' = p_1/(p_2/\ldots/p_m \uplus p'_2/\ldots/p'_n)$;
     . Otherwise, $p \uplus p' = [p, p']$.
   - Otherwise, $p \uplus p' = [p, p']$.

The purpose of *tree join* operator is used to union two path terms together to form one path term if these two path terms have the same URL. The common path between the two terms are merged into one path.

Definition 6.13. Let XDB be a set of positive ground absolute path expressions with $XDB \equiv \{(u_1)/p_1, \ldots, (u_m)/p_m\}$. Then *tree join* operation on XDB, denoted by $\uplus XDB$ is generated as for any two expressions u_i/p_i and u_j/p_j in XDB, if $u_i = u_j$, then u_j/p_j is removed and combined into u_i/p_i to form an expression as $u_i/(p_i \uplus p_j)$. When all expressions in XDB have different URLs, then the XDB is the result of *tree join* operation.

Definition 6.14. Let XDB be an XML database, R is a set of deductive rules defined in XDB. The *immediate deductive consequence* operator T_R over XDB is defined as follows,

$$T_R(XDB) = \uplus\{\theta H | H \text{ :- } L_1, \ldots, L_n \in R \text{ and } \exists \text{ a ground substitution } \theta \text{ such that } XDB \vDash \theta L_1, \ldots, XDB \vDash \theta L_n\}$$

The *immediate deductive consequence* operator is used to generate the derived properties from the deductive rules.

Definition 6.15. Let XDB be an XML database, R is a set of deductive rules defined in XDB. The *deductive completeness* of XDB, denoted by $XDB*$ is generated as follows,

$XDB* = XDB$

while $(T_R(XDB*) \not\preceq XDB*)$

$XDB* = \uplus((XDB*) \cup T_R(XDB*))$

The *deductive completeness* is generated using the immediate deductive consequence step by step until no new derived properties come out.

Theorem 3. Let XDB be an XML database and R is a set of rules defined in XDB. Then $XDB*$ is a minimal model of R.

Proof. First we prove $XDB*$ is a model of R. From the algorithm of generating $XDB*$, we know $T_R(XDB*) \preceq XDB*$. $\forall r \in R$, let $r \equiv H$:- $L_1, \ldots, L_n$. $\forall \theta$, suppose $XDB \vDash \theta L_1, \ldots, XDB \vDash \theta L_n$, then by definition 6.15, we know $T_R(XDB*) \vDash \theta H$. Since $T_R(XDB*) \preceq XDB*$, $XDB* \vDash \theta H$. Therefore $XDB* \vDash r$, and $XDB*$ is a model of R.

Suppose $XDB*$ is not a minimal model of R, then $\exists$ a model N such that $XDB* \not\preceq N$. Since $XDB*$ is generated by union with $T_R(XDB*)$, so $\exists \theta H \in T_R(XDB*)$ such that $N \not\models \theta H$. However, since N is a model, so $N \models \theta H$ and result in a contradiction. Therefore $XDB*$ is a minimal model. □

Intuitively, the deductive completeness $XDB*$ is extended with those new properties that are derived from the deductive rules and are necessary. No other properties are inside the $XDB*$. So the $XDB*$ is a minimal model.

Definition 6.16. Let XDB be an XML database and Q a query. Then the *semantics* of Q under XDB is given by $T_Q(XDB*)$ as follows,

$$T_Q(XDB*) = \uplus\{\theta A | A \Leftarrow L_1, \ldots, L_n \text{ and } \exists \text{ a ground substitution } \theta \text{ such that } XDB* \models \theta L_1, \ldots, XDB* \models \theta L_n\}$$

where $XDB*$ is a minimal model of R and R is the set of deductive rules used directly or indirectly in $L_1, \ldots, L_n$.

Given a query, we need to get those rules that are used by this query. Then a minimal model of these involved rules is generated using the bottom up approach similar to Datalog. Finally, the semantics of the query is trivially defined by a tree join operator.

Chapter 7

Comparison with Related Works

In this chapter, we will compare our XDO2 language with some other popular and powerful XML querying languages, such as XQuery [6] which is the current standard of W3C, and XTreeQuery [10] which is briefly introduced in section 2.1. We will also compare our XDO2 language with some other logical querying languages, such as F-logic [23], which combines the object-oriented paradigm and deductive paradigm elegantly, and a logical foundation for XML [31].

When we compare the XDO2 language with other languages, we will compare the expressive power and the features they support both in syntax and semantics level. This is because in syntax level, some features may not be supported while they are supported in semantics level for some languages. For example, the multi-valued variable feature is not supported in syntax level in XQuery. However, the multi-valued variable feature is supported in semantics level using the *LET* clause in XQuery.

The comparison is based on the expressive power of the query languages and the features they support both in syntax and semantics level. The following criteria are used:

1. Underlying data: what kind of data the querying supports.
2. Path expressions: how paths are specified in the query.
3. Deductive rule: whether the query language supports deductive rules as part of querying.
4. Negation: how to express the negation querying, using not-predicate or using the conventional logical negation.
5. Recursion: how to express recursion, using recursive rules, using recursive querying directly, or using recursive functions.
6. Quantification: whether it is necessary to use quantifiers to express universal and existential queries.
7. Multi-valued variable: whether the multi-valued variables are explicitly indicated or not.
8. Structure query: whether the query of the structure information is supported naturally or directly.
9. Object-oriented features: whether the query language support object-oriented features or not.

Table 7.1 and table 7.2 compare the expressive power and the features of five query languages, XDO2 query language, XQuery, XTreeQuery, F-logic, and a logical foundation for XML in both syntactic and semantic level.

	XDO2	**XQuery**	**XTree Query**	**F-logic**	**Logical foundation for XML**
Underlying data	XML tree	XML tree	XML tree	Object	XML tree
Path expression	XTree	XPath	XTree	Path expression	XTree-like expression
Deductive rule	Yes	No	No	Yes	Partial
Negation	not-predicate	logical negation	logical negation	logical negation	logical negation
Recursion	recursive rules	recursive function	recursive query	recursive rule	recursive query
Quantification	**No**	Yes	Yes	Yes	Yes
Multi-valued variable	Yes	**No**	Yes	**No**	Yes
Structure querying	Yes	**No**	Yes	Yes	Yes
Object-oriented features	Yes	No	No	Yes	No

Table 7.1: Syntactic comparison between XML query languages

From the table 7.1 and table 7.2, we can see that there are four different entry pairs highlighted in the comparison criteria of *Quantification*, *Multi-valued variable* and *Structure querying*. In syntax level speaking, XDO2 does not support quantification. However, XDO2 supports the meaning of quantification through an equivalent way of using not-predicate. For multi-valued variable issue, both the XQuery and F-logic do not support it syntactically. The multi-valued variables are supported using some special clauses or expressions. For the structure query issue, the XQuery can not support it directly. It needs to use some functions to support it.

In semantics level speaking, we can see that all the five languages support the quantification, multi-valued variable, and structure querying, which means they have the same expressive power or have the same features sup-

	XDO2	**XQuery**	**XTree Query**	**F-logic**	**Logical foundation for XML**
Underlying data	XML tree	XML tree	XML tree	Object	XML tree
Path expression	XTree	XPath	XTree	Path expression	XTree-like expression
Deductive rule	Yes	No	No	Yes	Partial
Negation	not-predicate	logical negation	logical negation	logical negation	logical negation
Recursion	recursive rules	recursive function	recursive query	recursive rule	recursive query
Quantification	**Yes**	Yes	Yes	Yes	Yes
Multi-valued variable	Yes	**Yes**	Yes	**Yes**	Yes
Structure querying	Yes	**Yes**	Yes	Yes	Yes
Object-oriented features	Yes	No	No	Yes	No

Table 7.2: Semantic comparison between XML query languages

ported. However, our XDO2 language supports these either in syntax level or through an equivalent simple way, which results in the XDO2 language is more simple and compact, and more convenient to use.

After discussing the difference between the syntax and semantics level, we will start to compare our XDO2 language with other query languages. The success of *F-logic* [23] was due to the clean combination of the object-oriented and deductive paradigms. However, the underlying data in F-logic are objects and can not handle the current popular XML tree data structure. The XDO2 language is designed for the XML tree data while including the deductive and object-oriented features.

The XML query languages, such as *XQuery* [6], and *XTreeQuery* [10] can not support the deductive rule which can be used to derive new properties

to simplify the querying as in XDO2. The XDO2 query language supports the recursive query more naturally by using the recursive deductive rules instead of by using recursive querying or by recursive functions. The *logical foundation* [31] for XML is a language with the deductive features, however it can not support object-oriented features like XDO2 can. Furthermore, since XDO2 is based on XTree, where queries are more compact, more convenient to write and understand than XPath queries, the XDO2 inherits these merits.

Another major difference between XDO2 and other query languages for XML lies in the use of the not-predicate [26] for querying. By using the not-predicate for querying, the variables inside the negation terms are existentially quantified. However, the negation querying in other query languages is achieved by using logical negation which needs the argument to be a boolean value. As a result, using the not-predicate, the universal and existential quantifiers can be avoided in XDO2 which can still achieve the same expressive power as those languages using the universal, existential and logical negation quantifiers. The queries using not-predicate are more simple and compact.

Finally, I would like to highlight again that our XDO2 query language is based on XTree and has all the advantages of XTree compared with XPath. The expressions based on XTree are more simple and compact, have a compact query return format, indicate multi-valued variables explicitly, and support structure querying naturally. All these features makes the XDO2 language to be more simple and compact, and more convenient to use.

Chapter 8

Conclusion and Future Works

In this chapter, we summarize our research work and what we have done in section 8.1. In addition, we point out some research directions that can be used for further research study in section 8.2.

8.1 Conclusion

Deductive database and object-oriented database are two extensions of the current relational database system. More recently, an XML query language XTree was proposed. Queries written in XTree are more compact, more convenient to write and easier to understand than queries written in XPath.

Guided by this, we propose a novel new XML query language XDO2 which is based on XTree and has deductive database features and object-oriented features. Our XDO2 language is more *compact*, and *convenient* to use than current query languages for XML such as XQuery or XPath because it is based on XTree, and supports deductive rules and not-predicate negation. It is also very *powerful* because of the recursive deductive rules. Some object-

oriented features are also supported.

In this thesis, we present some preliminary background on XTree, deductive databases, and object-oriented databases. An XDO2 database example is presented to motivate the discussion of the XDO2 language. The XDO2 language features from XTree, from object-oriented databases, and from deductive databases are explained in a more systematic and complete manner. The formal treatment of the language syntax and semantics are presented. Finally, a comparison of our XDO2 language with other XML query languages is presented. In addition, we also introduce how to define deductive rules, the relationship type of ORA-SS schema model, and superclass attribute in XML Schema in the appendices.

A summary of comparison of the XDO2 language with XQuery are:

1. Negation is supported in the XDO2 language with semantics similar to the not-predicate instead of the conventional logic negation symbol "$\sim$" which negates the boolean expression and is used in XQuery by "NOT". In XQuery, the logical negation "NOT" is usually used with "EVERY", "SOME" for those universal and existential queries. While XDO2 uses not-predicate alone and gains the same expressive power. A consequence of this decision is that XDO2 is able to support nested negation and negation of sub-trees naturally in a compact form. Although XQuery can also use multiple XPath expressions and quantified expressions to achieve the same expressive power, our XDO2 langauge will be more simple and compact.

2. Methods that deduce new properties are implemented as deductive rules. XDO2 can use the new properties directly. The presence of

recursive deductive rules makes the recursive querying possible. While in XQuery, it can use the functions to achieve the same thing, but it complicates the querying while our deductive rules are defined within the database.

3. Schema querying is made possible with a special term *stru : value* to explicitly distinguish the element tag (attribute name) from the element value (attribute value). *Stru* binds to the element tag (attribute name) and *value* binds to the element value (attribute value). Unlike in XQuery, the name and value pair are bound to the variables together. Therefore, in XQuery, in order to get the element tag (attribute name) or the value of the element (attribute), we have to use predefined built-in functions on the variables, such as *local-name* and *string* to get the element tag (attribute name) and value of the variables respectively.

4. Inheritance enables a subclass object can inherit all the attributes and sub-elements from its superclass objects. These inherited properties can be directly used in querying. However, XQuery currently does not support it.

5. Features such as the binding of multiple variables in one expression, compact return format and explicit multi-valued variables are supported in the XDO2 language naturally due to the influence of XTree. Since XQuery is based on XPath, it does not have these merits.

In summary, we have developed a more compact, convenient to use, and powerful XML query language with deductive rules, not-predicate negation, and the support of some object-oriented features.

8.2 Future Works

In the future, we would like to use the XDO2 query language to define views. Since our XDO2 query language has a simple compact result format for XML query result compared with nested queries or mixed data with queries as in XQuery, it is expected that an XDO2 view definition will be more compact and easier to understand compared with XQuery.

We would also like to investigate how to evaluate the XDO2 query language efficiently, using a Prolog like top-down approach, or Datalog like bottom-up approach, or combine both. It may be possible that some queries are efficient using a top-down approach while some queries are efficient using a bottom-up approach. How to handle the negation querying and recursive querying efficiently will be the major problem in the query evaluation procedure.

Bibliography

[1] S. Abiteboul and P.C. Kanellakis. Object identity as a query language. *Journal of ACM*, 45(5):798–842, 1998.

[2] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 32–41, 1993.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructure data. *International Journal of Digital Library*, 1(1):68–99, 1997.

[4] M.L. Barja, A.A.A. Fernandes, N.W. Paton, and M.H. Williams. Design and implementation of ROCK&ROLL: A deductive object-oriented database system. *Information System*, 20:185–211, 1995.

[5] P.V. Biron and A. Malhotra. XML schema part 2: Datatypes, May 2001. http://www.w3.org/TR/xmlschema-2.

[6] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. W3C Working Draft, July 2004. http://www.w3.org/TR/xquery.

[7] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yeergeau. Extensible markup language (XML) 1.0 (third edition), 2004. http://www.w3.org/TR/REC-xml.

[8] F. Cacace, S. Ceri, S. Crepi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modelling with a rule-based programming paradigm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 225–236, 1990.

[9] W. Chen and D. Warren. C-logic for complex objects. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 369–378, 1989.

[10] Z. Chen, T.W. Ling, M.C. Liu, and G. Dobbie. XTree for declarative XML querying. In *Proceedings of DASFAA*, pages 100–112, Korea, 2004.

[11] P. Chippimolchai, V. Wuwongse, and C. Anutariya. Semantic query formulation and evaluation for XML databases. In *Proceedings of WISE*, pages 205–214, Singapore, 2002.

[12] J. Clark. XSL transformations (XSLT) version 1.0, November 1999. http://www.w3.org/TR/xslt.

[13] J. Clark and S. DeRose. XML path language(XPath) version 1.0, November 2001. http://www.w3.org/TR/xpath.

[14] K.L. Clark. Negation as failure. In *Logic and Databases*, pages 293–322, 1977.

[15] A. Davidson, M. Fuchs, M. Hedin, M. Jain, J. Koistinen, C. Lloyd, M. Maloney, and K. Schwarzhof. Schema for object-oriented XML 2.0, July 1999. http://www.w3.org/TR/NOTE-SOX.

[16] G. Dobbie and R. Topor. On the declarative and procedural semantics of deductive object-oriented systems. *Journal of Intelligent Information System*, (2), 1995.

[17] G. Dobbie, X.Y. Wu, T.W. Ling, and M.L. Lee. ORA-SS: An object-relationship-attribute model for semistructured data. Technical Report TR21/00, School of Computing, National University of Singapore, 2000.

[18] D.H. Fishman et al. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, 1987.

[19] O. Deux et al. The story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[20] D.C. Fallside. XML schema part 0: Primer, May 2001. http://www.w3.org/TR/xmlschema-0.

[21] H.M. Jamil. Implementing abstract objects with inheritance in $datalog^{neg}$. In *Proceedings of the International Conference on Very Large Data Bases*, pages 46–65, 1997.

[22] P. Kandzia and C. Schlepphorst. Florid - a prototype for F-Logic. In *German Workshop on Logic Programming*, 1997.

[23] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42(4):741–843, 1995.

[24] W. Kim. *Introduction to object-oriented databases*. The MIT Press, Cambridge Massachusetts, 1990.

[25] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore system. *Communications of the ACM*, 34(10):50–63, 1991.

[26] T.W. Ling. The prolog not-predicate and negation as failure rule. *New Generation Computing*, 8(1):5–31, 1990.

[27] T.W. Ling and W.B.T. Lee. DO2: A deductive object-oriented database system. In *Proceedings of the 9th International Conference on Database and Expert System Applications*, pages 50–59, 1998.

[28] T.W. Ling and P.K. Teo. Inheritance conflicts in object-oriented systems. In *DEXA*, pages 189–200, 1993.

[29] M. Liu and T.W. Ling. Towards declarative XML querying. In *Proceedings of WISE*, pages 127–138, Singapore, 2002.

[30] M.C. Liu. The ROL deductive object base language. In *Proceedings of Database and Expert Systems Application*, pages 189–200, 1993.

[31] M.C. Liu. A logical foundation for XML. In *CAiSE*, pages 568–583, 2002.

[32] M.C. Liu, G. Dobbie, and T.W. Ling. A logical foundation for deductive object-oriented databases. *ACM Transactions Database Systems*, 27(1):117–151, 2002.

[33] Y. Lou and M. Ozsoyoglu. LLO: A deductive language with methods and method inheritance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 198–207, 1991.

[34] D. Maier. A logic for objects. Technical Report E-86-012, Oregon Graduate Institute, Beaverton, Oregon, 1986.

[35] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2000.

[36] J. Robie, J. Lapp, and D. Schach. XML query language (XQL), 1998. http://www.w3.org/Tands/QL/QL98/pp/xql.html.

[37] L.A. Rowe and M. Stonebraker. The postgres data model. In *VLDB*, pages 83–96, 1987.

[38] V. Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, Objectstore, O2. *SIGMOD Record*, 21(1):93–104, 1992.

[39] D. Srivastava, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Adding object-orientation to a logic database language. In *Proceedings of the International Conference on Very Large Data Bases*, pages 158–170, 1993.

[40] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures, May 2001. http://www.w3.org/TR/xmlschema-1.

[41] J.D. Ullman. *Principles of Database and Knowledge-base Systems*, volume I. Computer Science Press, 1988.

[42] G. Wang and M.C. Liu. Extending XML schema with nonmonotonic inheritance. In *Conceptual Modeling for Novel Application Domains*, pages 402–407, 2003.

Appendix A

XML Schema Extension

Several XML schema languages have been proposed, such as DTD [7], SOX [15], XML Schema [5, 20, 40] whhich are used to constrain and define a class of XML documents. Some object-oriented features, such as typing, inheritance and complex object are already supported and standardized by W3C. In [42], the XML Schema is extended to include polymorphism, overriding, blocking, multiple inheritances and conflict handling. Specifically, it supports class inheritance, multiple inheritance with conflict resolution mechanism, such as rename, blocking, and explicitly inheriting, polymorphism with polymorphic elements and polymorphic references. However, the inclusion of deductive rules into XML Schema definition is still missing in the literature. Another XML Schema extension of this section is to include the definition of relationship type semantics as in ORASS model. Finally, we define the superclass attribute, which is used to combine this object with its superclass object in XML Schema.

A.1 Deductive Rule in XML Schema

Figure A.1 shows the type definitions for the person object class in XML Schema. We distinguish the *object class* from *complex element* using the ID attribute. If the element has an ID attribute defined in XML Schema, we say this is an object class, such as person object class. If the element does not have ID attribute defined, while it contains attributes or sub-elements, we say it is a complex element, such as the address complex element. *Simple elements* are those elements without attributes or sub-elements, i.e., the type is a simple type, such as string, or IDREF type. These simple elements are usually the attributes of some object class.

In person object class of Figure A.1, there is an attribute pno which uniquely identifies a person. An object class spouse of person is also specified with a pno id reference to person in the schema. There are a name simple element which is the name of the person and an address complex element which keeps the address of the person. The address complex element contains street and city simple elements. There is an birthyear simple element with default value 2004 in the schema. The sex simple element of person is defined.

As an extension to the XML Schema definitions, there are two methods defined for the person object class. One is the *age* method and the other is the *bachelor* method which are encapsulated inside the person object class definition. The *age* method returns the age of a person given the person's birthyear. It defines the method name as *age*, return type as *integer*, and a rule with head and body. The rule specifies that if the birthyear of the person is $b and $a is 2004 minus $b, then $a is the age of the person. The *bachelor* method returns a true value if the person's sex is "Male", and the

```
<xs:complexType name="personType">
    <xs:attribute name="pno"
                type="xs:ID" use="required" />
    <xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="address" type="addressType" />
        <xs:element name="birthyear" type="xs:integer"
                        default="2004" />
        <xs:element name="sex" type="xs:string" />
        <xs:element name="spouse" type="spouseType" />
    </xs:sequence>
    <xs:method name="age" returnType="xs:integer">
        <xs:rule>
            <xs:head>age : $a</xs:head>
            <xs:body>birthyear : $b, $a=2004-$b</xs:body>
        </xs:rule>
    </xs:method>
    <xs:method name="bachelor" returnType="xs:boolean">
        <xs:rule>
            <xs:head>bachelor : true</xs:head>
            <xs:body>sex : "Male", not(spouse : $s)</xs:body>
        </xs:rule>
    </xs:method>
</xs:complexType>
<xs:complexType name="addressType">
    <xs:sequence>
        <xs:element name="street" type="xs:string" />
        <xs:element name="city" type="xs:string" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="spouseType">
    <xs:sequence>
        <xs:attribute name="pno" type="IDREF"
                target="personType" use="implied" />
    </xs:sequence>
</xs:complexType>
```

Figure A.1: Type definition for class person

person does not have any spouse. It defines method name as *bachelor*, return type as *boolean*, and a rule with head and body. The rule specifies that if the person's sex is "Male", and the person does not have any spouse, then the true value is returned for bachelor method. Notice both the methods are simplified with relative path to person class instead of absolute path staring from the root since the method is encapsulated inside the object class.

As we noticed in the method definition of the schema, the *<xs:method>* construct is used to specify the method of an object class. It includes the method's name and method return type. It also contains the method's rule definition, possibly with multiple rules, such as method ancestor in example 4.8. Each rule has a head and body part with the meaning that by substituting variables with values, if the body part are satisfied, then the head part is true. The syntax of method declaration in XML Schema is as follows,

<XS:METHOD NAME="method_name" RETURNTYPE="type">
  <XS:RULE>
    <XS:HEAD>exp</XS:HEAD>
    <XS:BODY>exp_1,...,exp_n</XS:BODY>
  </XS:RULE>
  ⋮
</XS:METHOD>

The method_name is a name of the method. Type is one of the build-in types or user defined types. Exp, exp_1, ..., and exp_n are relative path expressions under the object class that are explained in chapter 5.

```
<xs:relationshipType name="ps">
    <xs:classsequence>
        <xs:element name="part" type="partType"
                minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="supplier" type="supplierType"
                minOccurs="0" maxOccurs="unbounded" />
    </xs:classsequence>
    <xs:sequence>
        <xs:element name="price" type="xs:integer" />
    </xs:sequence>
</xs:relationshipType>
```

Figure A.2: Type definition for relationship type ps

A.2 Relationship Type in XML Schema

Figure A.2 shows the type definition for relationship type ps. The binary relationship type ps defines the relationship between part and supplier object class. The participating constraint of part is 0:n, which means each part can be supplied by zero supplier, one supplier or more suppliers. The participating constraint of supplier is also 0:n, which means each supplier can supply zero part, one part or more parts. The simple element price is defined under the relationship type.

To extend the relationship type definition of ORASS schema model in XML Schema, we need to add the *<xs:relationshipType>* construct to specify it is a relationship definition instead of an element or attribute definition. It contains the participating object classes as well as their participating constraints and the elements that belong to the relationship type instead of some object class. The syntax of relationship type definition is as follows,

```
<XS:RELATIONSHIPTYPE NAME="relationship_name">
    <XS:CLASSSEQUENCE>
    <XS:ELEMENT NAME="class_name_1" TYPE="class_type_1" />
        ⋮
    <XS:ELEMENT NAME="class_name_d" TYPE="class_type_d" />
    </XS:CLASSSEQUENCE>
    <XS:SEQUENCE>
        <XS:ELEMENT NAME="element_name_1" TYPE="type_1" />
        ⋮
        <XS:ELEMENT NAME="element_name_n" TYPE="type_n" />
    </XS:SEQUENCE>
</XS:RELATIONSHIPTYPE>
```

In the <*xs:classsequence*> construct, the class names *class_name_1*, ..., *class_name_d* are the names of the object classes that participating in the relationship type, such as *part* and *supplier* in Figure A.2. And the class types *class_type_1*, ..., *class_name_d* are the types of the object classes, such as *partType* and *supplierType* in Figure A.2. In the <*xs:sequence*> construct, these element definitions are defined under the relationship type. Therefore, these elements belongs to the relationship type instead of the object classes.

A.3 Superclass Attribute in XML Schema

Figure A.3 shows the type definition for the subclass employee whose superclass is person. As we can see, the XML Schema use <*xs:extension*> construct with *bases* attribute to specify the superclass types. The attribute *eno* is defined as an ID for the employee class. The subelements *salary* and

```
<xs:complexType name="employeeType">
    <xs:extension bases="personType">
    <xs:attribute name="eno" type="xs:ID" use="required"/>
    <xs:attribute name="spouse" type="IDREF"
                target="personType" use="implied"/>
    <xs:attribute name="pno" type="SUPERCLASS"
                target="personType" use="required"/>
    <xs:sequence>
        <xs:element name="salary" type="xs:integer"/>
        <xs:element name="hobby" type="xs:string"
                minOccurs="1" maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
```

Figure A.3: Type definition for subclass employee with superclass person

hobby are also defined in the employee class. The most interesting part is the attribute *pno* definition which is used to refer the person object class. It is different from the normal IDREF attribute. The IDREF attribute refers to some object id and this attribute value is the object id. While superclass attribute also refers to some object id but this attribute is used to combine the two objects. In Figure A.3, the employee's IDREF attribute spouse means this employee object has spouse relationship with the person object identified by the oid value of spouse. But the SUPERCLASS attribute pno means this employee object is also a person object identified by the oid value of pno. The syntax used to define the superclass attribute is as follows,

<XS:ATTRIBUTE NAME=“attribute_name” TYPE=“SUPERCLASS”
TARGET=“class_type” USE=“required” />

The *attribute_name* is some attribute name and the *class_type* is some

base class that this class extends. *SUPERCLASS* is used as the keyword for superclass attribute similar to IDREF.