

DATALOG++: AN OBJECT-ORIENTED FRONT-END FOR THE XSB
DEDUCTIVE DATABASE MANAGEMENT SYSTEM

By

Zhixin Tang

A Project Report
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science

Mississippi State, Mississippi

May 1999

DATALOG++: AN OBJECT-ORIENTED FRONT-END FOR THE XSB
DEDUCTIVE DATABASE MANAGEMENT SYSTEM

By

Zhixin Tang

Approved:

Dr. Hasan Jamil
Assistant Professor of Computer Science
(Major Professor)

Dr. Susan Bridges
Associate Professor and Graduate
Coordinator of the Department of
Computer Science
(Committee Member)

Dr. Nancy Miller
Associate Professor of Computer Science
(Committee Member)

Name: Zhixin Tang

Data of Degree: May 13, 1999

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Hasan Jamil

Title of Study: DATALOG++: AN OBJECT-ORIENTED FRONT-END FOR THE XSB
DEDUCTIVE DATABASE MANAGEMENT SYSTEM

Pages in Study: 51

Candidate for Degree of Master of Science

As a deductive object oriented database language, Datalog++ nicely incorporates an object model with the deductive mechanism of Datalog. The object model supports most of the salient object-oriented features such as encapsulation, inheritance with overriding, conflict resolution of multiple inheritance, and access control of methods. The purpose of this project was to design and implement a Datalog++ front-end over an existing deductive database management system, XSB. The Datalog++ front-end has been successfully implemented on the Windows NT platform using Visual C++ and MFC. The front-end translates user programs from the Datalog++ language to the back-end XSB language and checks syntax and semantic errors during the translation. It supports interactive database operations such as loading schema and objects into a database, browsing the schema in the current database, and querying the database using consistent and familiar graphical user interfaces.

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my advisor, Dr. Hasan Jamil, for his guidance and support throughout my graduate program. Sincere thanks are also due to Dr. Susan Bridges and Dr. Nancy Miller for serving as my committee members.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	ii
LIST OF TABLES	v
LIST OF FIGURES	vi
 CHAPTER	
1. INTRODUCTION	1
2. BACKGROUND AND PROJECT DESCRIPTION	3
Deductive Database	3
Deductive Object-oriented Database (DOOD)	5
Comparisons of DOOD Languages.....	6
Project Objectives and Requirements	14
3. DATALOG++ FRONT-END	17
Front-end Datalog++ Language Specification	17
Schema Syntax	18
Object Syntax	20
Query Syntax	21
XSB Binding of the Datalog++ Rulebase	22
Front-end System Design	25
Front-end System Implementation	26
XSB Module	27
Compiler Module	27
User Interface Module	29
4. EVALUATION AND SUMMARY	31
Evaluation.....	31
Summary and Future Directions.....	32

REFERENCES	34
APPENDIX	
A. PROJECT CONTRACT	36
B. LIST OF PROJECT DELIVERABLES.....	40
C. SOME SNAPSHOTS OF THE FRONT-END	41
D. SOME EXAMPLE TEST CASES	44

LIST OF TABLES

TABLE	Page
2.1 Relation Parenthood	3
3.1 EBNF Representation of Schema Syntax	18
3.2 An Example Schema Syntax	19
3.3 EBNF Representation of Object Syntax	20
3.4 An Example Object Syntax	21
3.5 EBNF Representation of Query Syntax	22

LIST OF FIGURES

FIGURE	Page
2.1 An Example Database Program	7
2.2 Datalog++ Code Listing	8
2.3 FLORID Code Listing	9
2.4 Coral++ Code Listing	9
2.5 Rock & Roll Code Listing	10
3.1 XSB Binding of the Datalog++ Rulebase	23
3.2 Front-end System Architecture	25

CHAPTER 1

INTRODUCTION

For the past decade, deductive databases and object-oriented databases have been two parallel research and development focuses to extend the traditional relational database model (Jamil 1997). Deductive databases incorporate inference mechanism into relational database systems and handle complex recursive queries elegantly. On the other hand, object-oriented databases bring complex objects, encapsulation, and inheritance into the database world and are most appropriate for applications with complex data modeling requirements (e.g., geographical information systems).

Deductive and object-oriented database technologies have their own advantages and disadvantages. Deductive databases are based on first order logic. They support expressive declarative query languages with inference capability. Query processing and optimization in deductive databases have been extensively studied. However, data modeling facilities are limited in deductive databases. Object-oriented databases originate from object-oriented programming and are based on powerful object-oriented data models. Encapsulation and inheritance with overriding are two of the most important features of object-orientation. However, current object-oriented databases largely depend on navigational access to data from object-oriented programming languages such as C++, Java, and Smalltalk. Object-oriented databases lack a formal foundation and adequate

declarative query facilities. Since deductive databases and object-oriented databases are complementary in features, deductive object-oriented database (DOOD) appears to be a promising database technology to exploit both inference mechanism and object-orientation.

Datalog++ (Jamil 1997) is an object-oriented extension to Datalog. Datalog++ incorporates most of the salient object-oriented features, such as encapsulation with access control, inheritance with overriding, and multiple inheritance conflict resolution, in a pure logic framework. In Datalog++, users can write classes and objects to model their application domains. A reduction algorithm is used to rewrite classes and objects into logic clauses. This translation-based strategy is practical and efficient. Programs written in Datalog++ can be readily translated into a language that a back-end deductive database system understands.

This report discusses design and implementation of a Datalog++ front-end over an existing XSB deductive database system (Sagonas et al. 1998). The rest of this report is organized as follows. Chapter 2 gives background on deductive databases and deductive object-oriented databases and lays out the project objectives and requirements. Chapter 3 discusses several design and implementation issues of the Datalog++ front-end. The last chapter evaluates and summarizes the project.

CHAPTER 2

BACKGROUND AND PROJECT DESCRIPTION

This chapter is organized as follows. The Datalog language and the XSB deductive database system are introduced first. The second section discusses several typical DOOD languages and explains why Datalog++ is a better solution. The last section discusses the objectives of the project and functionalities of the intended front-end system.

Table 2.1
Relation Parenthood

Child	Parent
mary	Steve
john	Steve
mary	Ann
john	Ann
steve	Anderson
anderson	Jimmy
jimmy	Bob
bob	Diana
diana	Jones

Deductive Database

Datalog (Zaniolo et al. 1997) is a logic-based deductive database language. In Datalog, a database is represented as a set of facts, which are predicates with no variable

as arguments. A fact corresponds to a tuple of a table in the relational model. For example, the relation `parenthood` in Table 2.1 can be represented in Datalog by the following facts:

```
parenthood(mary,steve).
parenthood(john,steve).
parenthood(mary,ann).
parenthood(john,ann).
parenthood(steve,anderson).
parenthood(anderson,jimmy).
parenthood(jimmy,bob).
parenthood(bob,diana).
parenthood(diana,jones).
```

The obvious advantage of Datalog over SQL is its built-in inference mechanism.

In Datalog, handling of recursive queries becomes the DBMS's responsibility. For example, in SQL, if a query "Who are ancestors of john?" is intended, the user has to join the relation `parenthood` multiple times to retrieve all ancestors of john. How many times the relation has to be joined is unknown without looking at the contents of the relation. The proposed SQL3 is expected to address recursive queries like this. However, in Datalog, handling such recursive queries is just like handling non-recursive ones. This is because Datalog allows rules. In the above example, the following rules can be added to the deductive database to define the predicate `ancestor/2`:

```
ancestor(X,Y) :- parenthood(X,Y).
ancestor(X,Y) :- parenthood(X,Z), ancestor(Z,Y).
```

As a convention in logic languages, variables start with an upper-case letter or an underscore and constants start with a lower-case letter. A simple query " :- ancestor(john, X)." can be asked by the user and the system will automatically bind the variable X to

john's ancestors. Datalog can provides this elegant recursive query support due to its first order logic based inference mechanism.

A deductive database includes a set of facts and a set of rules. The set of facts forms the extensional part of the database (EDB). By applying the set of rules to the EDB, new facts may be derived. These derived facts belong to the intentional part of the database (IDB). The union of the EDB and the IDB is the model of the database. Queries are evaluated against the model.

XSB is a deductive database system developed by Sagonas et al. (1998). It supports all features of Datalog and it also has many advanced features. However, XSB is a Prolog-style logic programming system that uses top-down evaluation. Standard Prolog uses a depth-first search strategy for top-down clause resolution and is thus susceptible to getting into an infinite loop. However, thanks to its tabling technique (Chen and Warren 1996), XSB does not have such a problem. In this project, XSB acts as the back-end database engine. The decision to choose XSB was based on its source code availability. Its advanced features also reserve more room for future enhancement of the front-end system.

Deductive Object-oriented Database (DOOD)

Deductive object-oriented databases seek to integrate deductive and object-oriented paradigms in the context of databases. A deductive object-oriented database has an object model and supports object-oriented features. It also has built-in inference mechanism and supports declarative programming. Object-orientation and declarative

support are two principal criteria to evaluate deductive object-oriented database management systems and languages.

Over the past several years, a significant number of DOOD languages has been proposed and/or implemented. These languages include Orlog (Jamil and Lakshmanan 1992), Datalog++ (Jamil 1997), Coral++ (Srivastava et al. 1993), Rock & Roll (Barja et al. 1995), F-Logic (Kifer and Lausen 1989), FLORID (Frohn et al. 1997), ROL (Liu 1996), and others. Generally speaking, they fall into three categories:

1. Direct semantics for object-oriented features: for example, F-Logic, FLORID, Orlog, and ROL.
2. Indirect semantics by rewriting object-oriented constructs to a deductive database language: for example, Datalog++.
3. Indirect semantics by rewriting object-oriented constructs to a procedural language: for example, Coral++ and Rock & Roll.

Datalog++ is a pure logic based deductive object-oriented database language. We are using Datalog++ as the front-end DOOD language in this project. In the next section, we compare Datalog++ with FLORID, Coral++, and Rock & Roll.

Comparisons of DOOD Languages

Datalog++ is compared with FLORID, Coral++, and Rock & Roll in this section. Since encapsulation and inheritance with overriding are two of the most important features of the object-oriented paradigm, we will look at whether they are captured in

these languages and how they are captured. We will also look at declarative support of these languages.

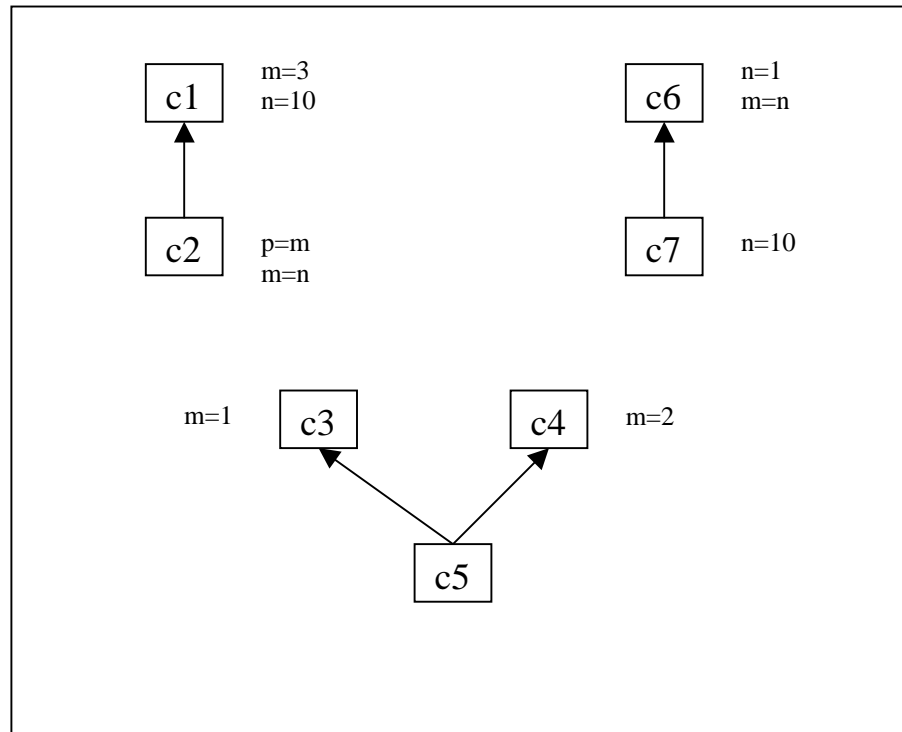


Figure 2.1 An Example Database Program

To bring out the differences, an example program is implemented in these languages. This example program is shown conceptually in Figure 2.1. Rectangles represent classes and arrows point from subclasses to superclasses.

Some questions to answer when comparing these languages are: How are attributes "m=3" and "n=10" inherited from c1 to c2? What is the value of m in c5 in a

multiple inheritance situation? Can a method be encapsulated in a class? How is the method "m=n" in c6 inherited to c7?

<pre>// schema part CLASS c1 { instance signatures { pub,val m/1; pub,val n/1;} } CLASS c2 subclass of {c1} { instance signatures { pub,val p/1;} } CLASS c3 { instance signatures { pub,val m/1;} } CLASS c4 { instance signatures { pub,val m/1;} }</pre>	<pre>CLASS c5 subclass of {c3,c4} { } CLASS c6 { instance signatures { pub,val n/1; pub,code m/1;} } CLASS c7 subclass of {c6} { } // object part c1.m(3); c1.n(10); c2.p(X) :- m(X); c2.m(X) :- n(X); c3.m(1); c4.m(2); c6.n(1); c6.m(X) :- n(X); c7.n(10);</pre>
---	---

Figure 2.2 Datalog++ Code Listing

Implementations of the example program in Datalog++, FLORID, Coral++, and Rock & Roll are shown in Figure 2.2, 2.3, 2.4, and 2.5, respectively. Familiarity with C++ and logic programming notations is assumed.


```

// * means inheritable
c1[m*->3; n*->10].
c2::c1.
c2[p->X] :- c2[m->X].
c2[m->X] :- c2[n->X].

c3[m*->1].
c4[m*->2].
c5::c3.
c5::c4.

c6[n*->1].
c6[m*->X] :- c6[n->X].
c7::c6.
c7[n->10].

```

Figure 2.3. FLORID Code Listing

<pre> // classes defined in C++ class c1 { public: int m; int n; c1() {m=3; n=10;} } class c2 : public c1 { public: int p; c2() {p=m; m=n;} } class c3 { public: int m; c3() {m=1;} } class c4 { public: int m; c3() {m=2;} } </pre>	<pre> class c5 : public c3, public c4 { } class c6 { public: int n; c3() {n=1;} int m() {return n} } class c7 { public: c7() {n=10;} } // rules defined in Coral p(c2,X) :- c2(P), X=P->p. m(c2,X) :- c2(P), X=P->m. m(c5, X) :- c3(P), X=P->m. m(c7, X) :- c7(P), X=p->m(). </pre>
--	--

Figure 2.4. Coral++ Code Listing

<pre> type c1: properties: public: m:integer, n:integer; ROCK: new (); end-type type c2: specialises: c1; properties: public: p:integer; ROCK: new (); ROLL: p_value(int), m_value(int); end-type type c3: properties: public: m:integer; ROCK: new (); end-type type c4: properties: public: m:integer; ROCK: new (); end-type type c5: specialises: c3, c4; ROLL: m_value(int); end-type type c6: properties: public: n:integer; ROCK: new (); mV() : int; end-type type c7: specialises: c6; ROCK: new (); ROLL: m_value(int); end-type </pre>	<pre> class c1 public: new () begin m=3; n=10;end end-class class c2 public: new () begin p=m; m=n;end p_value(int) begin p_value(X)@c2 :- X=p@c2; end m_value(int) begin m_value(X)@c2 :- X=m@c2;end end-class class c3 public: new () begin m=1;end end-class class c4 public: new () begin m=2; end end-class class c5 public: m_value(int) begin m_value(X)@c5 :- X=m@c3; end end-class class c6 public: new () begin n=1; end mV() begin return n; end end-class class c7 public: new () begin n=10; end m_value(int) begin m_value(X)@c7 :- X=mV()@c7; end end-class </pre>
---	---

Figure 2.5 Rock & Roll Code Listing

Both Datalog++ and FLORID are logic-based languages. The Datalog++ program has a schema part and an object part. In the schema, signatures of methods are declared. A signature represents attributes of the method. The signature specifies access control (public or private), inheritance mode (value inheritance or code inheritance), and the number of arguments the method expects. Signatures can be thought as function prototypes of C++. However, signatures do not specify types for arguments. In Datalog++, a method can be implemented only if an appropriate signature exists. FLORID does not have a schema part. Signatures are optional and the language does not check methods against signatures.

Unlike Datalog++ and FLORID, Coral++ and Rock & Roll are integrations of two languages. Coral++ has two sublanguages C++ and Coral. Classes/objects are defined and manipulated by the procedural sublanguage C++ and type checking is done by the C++ compiler. The sublanguage Coral is logic-based and used to query the database. C++ methods can be called in a rule or a query of Coral. For example, in the rule " $p(c2, X) :- c2(P), X=P \rightarrow p.$ ", variable P binds to an object of class c2 and $P \rightarrow p$ accesses the attribute p of the object. The rule intends to find out the value of p in the c2 class. However, in order to recognize method calls to the C++ module, the Coral deductive system has to be compiled with the C++ part of the program. This is a serious drawback of Coral++ because any modification to classes/objects will require recompilation of the whole system. Similarly, Rock & Roll has a procedural sublanguage Rock and a logic language Roll. Rock & Roll is a better integration because both sublanguages are built on the same

underlying object model. Unlike C++, Rock is an interpretive language. It does not compile to native code. Instead it compiles to an intermediate representation that is interpreted. This prevents conventional compile/link cycle and thus no recompilation is required. It does not have the drawback discussed above for Coral++. The common disadvantage of Coral++ and Rock & Roll is that they lack a formal foundation and data independence. The user has to implement methods in a procedural programming language and in the implementation navigational access to data is used. Once database schema or indexing changes, the implementation has to be modified.

Let us find out what the values are for m , n , and p in the class $c2$. In the class $c1$, m and n are attributes because they are constant. This case is to test how attributes are inherited. In Datalog++, inheritability is decided statically. Since m is redefined in the class $c2$, overriding takes priority and m is not inherited. But n is inherited. So in the class $c2$, $n=10$, $m=10$, and $p=10$. In FLORID, inheritability is decided dynamically. FLORID uses a bottom-up fixpoint evaluation strategy (Frohn et al. 1997). It deduces new facts from already established facts using a forward chaining technique. In program evaluation, logic rules are given priority over inheritance. After reaching a fixpoint by applying logic rules to deduce new facts (Tp operator), FLORID tries to deduce a new fact by inheritance. When there are several possible facts inheritable at the same time, FLORID chooses one of them non-deterministically. In our program, if m is inherited first, the answer is $m=3$, $n=10$, and $p=3$; if n is inherited first, then $m=10$, $n=10$, and $p=10$. Which one should be the right answer? Non-determinism of inheritance is a problem of

FLORID. In Coral++, inheritance is static. The answer is $m=10$, $n=10$, and $p=3$. Notice if we change the order of "p=m;" and "m=n;" in the c2 constructor, the answer will be $m=10$, $n=10$, and $p=10$. The order matters in a procedural language. In Rock & Roll, the answer is $m=10$, $n=10$, and $p=3$ because Rock has inheritance properties similar to C++.

In a multiple inheritance situation, if an inheritance ambiguity occurs, how is inheritance handled? In Datalog++, the method is not inherited. Thus in the class c5, m is undefined. FLORID inherits one of the choices non-deterministically. In the class c5, m can be 1 or 2. In Coral++ and Rock & Roll, the user should cast the class to one of its superclasses to resolve the ambiguity. Otherwise it is a run-time error. We argue that Datalog++ approach to multiple inheritance ambiguity is better because of its determinism. Datalog++ also provides means for the user to resolve the ambiguity by allowing the user to specify that inheritance of a signature or method is rejected from a superclass.

All these four languages support encapsulation because methods can be defined for classes. However, in FLORID, methods themselves are not inheritable. Instead, the result of the method application in the class can be inherited to subclasses/instances. Datalog++ supports both value inheritance and code inheritance. By value inheritance, the method is evaluated in the superclass and the result is inherited to the subclass. By code inheritance, the code of the method is inherited to the subclass and evaluated in the subclass. Coral++ follows the C++ inheritance rules. In Rock & Roll, Rock methods have inheritance rules similar to the inheritance rules of C++. Let us look at the value of m in

the class *c7*. In the *Datalog++* program, code inheritance is specified in the schema, so $m=10$. Value inheritance would make $m=1$. In *FLORID*, m is 1 in the class *c7* because m is 1 in *c6* and this value is inherited. Both *Coral++* and *Rock & Roll* will have 10 as the value of m in *c7*. *FLORID* does not support access control to methods while *Datalog++*, *Coral++*, *Rock & Roll* do.

In summary, *Coral++* and *Rock & Roll* does not have declarative semantics because of their procedural sublanguages. No matter how good the integration is, the user has to implemented classes and methods using procedural programming languages. Navigational access to data in the database has to be used and is labor-intensive and error-prone. *Coral++* and *Rock & Roll* thus do not promote data independence. On the other hand, both *Datalog++* and *FLORID* are logic-based and thus have declarative semantics. We argue that declarative access is a fundamental concept of database technology and should also be supported by deductive object-oriented databases. With respect to object-oriented modeling, *Datalog++* captures encapsulation and inheritance with overriding better than *FLORID* and provides richer object-orientation support. Dynamic and non-deterministic inheritance in *FLORID* is problematic while static and deterministic inheritance is intuitive. *Datalog++* supports both value and code inheritance. It also supports access controls to methods.

Project Objectives and Requirements

This project is to implement an object-oriented *Datalog++* front-end for the *XSB* deductive database system on the Windows NT platform.

From the previous section, we see that object-oriented features such as encapsulation, methods, access control, inheritance with overriding, and multiple inheritance can be incorporated into deductive databases in a clean declarative way. However, manually implementing these object-oriented features in a deductive database system such as XSB would be tedious and error-prone. It would be nice to provide the user an object-oriented front-end that allows him/her to think and code in an object-oriented way. The front-end transparently translates the user's program into whatever the back-end deductive system understands. This is one main objective of this project. The other objective is to build a prototype system for the Datalog++ language.

The front-end should provide a complete database programming environment. Most of today's DOOD prototypes (e.g., Coral++ and Rock & Roll) only support text-based interfaces. However, to simplify the users' job, a graphical user interface (GUI) should be used in this prototype of Datalog++. The following are the functional requirements of the front-end system:

1. The system should provide an editor for users to edit schema and object definitions.
2. The system should parse schema and object definitions, and translate them and submit them to the XSB back-end in an interactive fashion.
3. The system should include a schema analyzer to check well-typedness of object method definitions and to provide class information when requested by users.

4. The system should provide a query interface for users to query the deductive database.
5. The system should display query responses in a user-friendly manner.

CHAPTER 3

DATALOG++ FRONT-END

We've implemented a Datalog++ front-end for the XSB deductive database system . In this chapter, we discuss several design and implementation issues, such as the front-end language specification, translator/compiler construction, the XSB mapping of the Datalog++ rulebase, and system architecture considerations.

Front-end Datalog++ Language Specification

Theoretical treatment of Datalog++ is given in Jamil's paper (Jamil 1997). For the purpose of implementation, a detailed specification is given in the EBNF notation in this section. The specified front-end language can be treated as a subset of Datalog++.

When a relational database is built using SQL, the process usually involves defining the database schema, instantiating the database, and finally querying the database. The database schema defines the structure and other constraints of the database and a database instance is the current snapshot of the database data. The Datalog++ front-end follows this schema-data-query order of SQL. Naturally, Datalog++ syntax is divided into schema syntax, object syntax, and query syntax. Breaking Datalog++ syntax into components was also an attempt to facilitate a modular design.

Table 3.1
EBNF Representation of Schema Syntax

```

schema : ( class )*;
class : "CLASS" class_id subdecl classbody;
subdecl : { "subclass" "of" "{" sublist "}" };
sublist : class_id ( "," class_id )*;
classbody : "{" { class_sig } { ins_sig } { controls } "}";
class_sig : "class" "signatures" "{" sig_body "}";
sig_body : ( method_decl )*;
method_decl : pub_priv "," code_val meth_id "/" arity ";";
pub_priv : "pub" | "priv";
code_val : "code" | "val";
ins_sig : "instance" "signatures" "{" sig_body "}";
controls : "controls" "{" control_body "}";
control_body : ( reject_decl )*;
reject_decl : "reject" sig_meth value_id "/" "from" class_id ";";
sig_meth : "sig" | "meth";
class_id : value_id;
meth_id : value_id;
value_id : "[a-z][a-zA-Z0-9]*";
arity : "[0-9]+";

```

Schema Syntax

The schema syntax is described formally in Table 3.1 using the EBNF notation (Parr 1996). Terminals are quoted regular expressions and non-terminals are in lower case. Braces enclose optional elements. The symbol | is used to specify alternative rules. To make the syntax more concrete, we give a simple example in Table 3.2. In this example, the class gta has superclasses grad_stud and faculty (assume grad_stud and faculty have been defined). It defines method signatures class_name/1 and taship/1 locally. Class method signatures are inherited to subclasses but not to instances while

instance signatures are inherited to both subclasses and instances. Methods can be public (pub) or private (priv). Methods are either code inheritable (code) or value inheritable (val). In code inheritance, the actual method code is inherited to subclasses/instances and gets evaluated in the subclasses/instances. In value inheritance, the method is evaluated and the result is inherited to subclasses/instances. In a class definition, signatures (sig) or methods (meth) can be rejected from superclasses to provide a mechanism for users to resolve ambiguity in multiple inheritance. The class gta rejects signatures salary/1 and income/1 from its superclass faculty (assume signatures salary/1 and income/1 are defined in faculty).

Table 3.2
An Example Schema Syntax

```

CLASS gta subclass of {grad_stud, faculty}
{
  class signatures
  {
    pub, code class_name/1;
  }

  instance signatures
  {
    priv, val taship/1;
  }

  controls
  {
    reject sig salary/1 from faculty;
    reject sig income/1 from faculty;
  }
}

```

Object Syntax

The EBNF representation of object syntax is given in Table 3.3. In the object syntax, instances are declared; methods are defined for class objects and instance objects; and relationships between objects are specified using global predicates. An example object syntax is given in Table 3.4. In the example, joe and mary are both instances of

Table 3.3
EBNF Representation of Object Syntax

```

program : (clause)+;
clause : global_clause | local_clause | ins_clause |special_clause";" ;
global_clause : global_pred { ":-" clause_body };
local_clause : local_pred { ":-" clause_body };
ins_clause : ins_pred { ":-" clause_body };
special_clause : "hilog" pred_id;
global_pred : pred_id {args};
local_pred : obj_id "." global_pred;
ins_pred : obj_id ":" class_id;
var_isa_pred : (var | obj_id) (":" | "::") (var | obj_id);
mesg_pred : (obj_id | var) "<<" global_pred;
clause_body: (global_pred | mesg_pred | var_isa_pred |system_pred) (","
(global_pred | mesg_pred | var_isa_pred|system_pred))*;
system_pred : exp ("is" | "=" | "\=" | "<" | ">" | ">=" | "=<") exp;
exp : basic { ("+" | "-" | "*" | "/" ) exp1};
exp1: basic | "(" exp ")";
basic : float | var;
args : "(" term ( "," term)* ")";
term : value_id | quoted_id | float | var;
pred_id: value_id;
obj_id : value_id;
class_id: value_id;
value_id : "[a-z][a-zA-Z0-9_]*";
quoted_id "'~[~]+'"
float: "{\-}[0-9]+{.[0-9]+}";
var: "[A-Z_][a-zA-Z0-9_]*";

```

class `grad_stud` (assume defined in schema). Method `stipend/1` is defined for class `grad_stud` and specifies that a graduate student's stipend defaults to 12000. Method `stipend/1` is redefined in the instance `joe`, saying `joe` has a stipend of 2000 instead the default 12000. The last clause defines a relationship and means that `joe` and `mary` are classmates.

The current version of the front-end `Datalog++` supports the following system defined predicates: arithmetic predicates (`+`, `-`, `*`, and `/`), numeric comparison predicates (`=`, `\=`, `>`, `<`, `>=`, and `=<`), and an assignment predicate (`is`). To exploit aggregate features of the back-end `XSB`, the front-end also support a special `hilog` clause, which specifies a predicate symbol as a `hilog` term (see `XSB` for details).

Table 3.4
An Example Object Syntax

```
grad_stud.stipend(12000);
joe : grad_stud;
joe.stipend(20000);
mary : grad_stud;
classmate(joe, mary);
```

Query Syntax

The EBNF representation of query syntax is given in Figure 3.5. Query syntax is just the `clause_body` part of object syntax. For example, query `":- joe<<stipend(X);"` is intended to ask how much `joe`'s stipend is.

Table 3.5
EBNF Representation of Query Syntax

```

query : ":-" clause_body ";";
clause_body: (global_pred | mesg_pred | var_isa_pred | system_pred)
(", " (global_pred | mesg_pred | var_isa_pred | system_pred))*;
global_pred : pred_id {args};
var_isa_pred : (var | obj_id) (":" | "::") (var | obj_id);
mesg_pred : (obj_id | var) "<<" global_pred;
system_pred : exp ("is" | "=" | "\=" | "<" | ">" | ">=" | "=<") exp;
exp : basic { ("+" | "-" | "*" | "/" ) exp1 };
exp1: basic | "(" exp ")";
basic : float | var;
args : "(" term ( "," term)* ")";
term : value_id | quoted_id | float | var;
pred_id: value_id;
obj_id : value_id;
value_id : "[a-z][a-zA-Z0-9_]*";
quoted_id "'~[']+"
float: "{\-}[0-9]+{.[0-9]+}";
var: "[A-Z_][a-zA-Z0-9_]*";

```

XSB Binding of the Datalog++ Rulebase

The reduction algorithm to translate Datalog++ constructs to pure logic clauses is discussed in Jamils's paper (1997). In order to enforce the object model of Datalog++, a rulebase needs to be loaded into the back-end deductive database. A mapping of this rulebase to the Coral deductive database system is given in the paper (Jamil 1997). Here we give the XSB binding of the rulebase in Figure 3.1. Basically the rulebase defines predicates `sig_can_inherit/7`, `meth_can_inherit/5`, and `vis/4` to enforce rules of signature inheritance, method inheritance, and visibility (access control), respectively. All predicates in the rulebase should be protected from being redefined in the users'

```

%% isa_rules_start
tins(X,Y) :- ins(X,Y).
tins(X,Y) :- ins(X,Z), tsub(Z,Y), not (ins(X,Y)).
tsub(X,X) :- class(X).
tsub(X,Y) :- sub(X,Y).
tsub(X,Y) :- sub(X,Z), tsub(Z,Y).
tisa(X,Y) :- tins(X,Y).
tisa(X,Y) :- tsub(X,Y).
%% isa_rules_end

%% signature_inheritability_rules_start
inherit_sig(M_name,Ar,Vis,M_mode,Level,Obj,Obj,Obj) :-
    sig(Obj,Vis,M_mode,M_name,Ar,Level).
inherit_sig(M_name,Ar,Vis,M_mode,ins,Obj,S_obj,Sou_obj) :-ins(Obj,S_obj),
    tsub(S_obj,SS_obj),
    inherit_sig(M_name,Ar,Vis,M_mode,ins,S_obj,SS_obj,Sou_obj).
inherit_sig(M_name,Ar,Vis,M_mode,Level,Obj,S_obj,Sou_obj) :- sub(Obj,S_obj),
    tsub(S_obj,SS_obj),
    inherit_sig(M_name,Ar,Vis,M_mode,Level,S_obj,SS_obj,Sou_obj),
    not(rej(sig,M_name,Ar,Obj,S_obj)), not(sig(Obj,_,_,M_name,Ar,_)).
conflict_sig(M_name,Ar,Obj) :-
    inherit_sig(M_name,Ar,_,_,Obj,_,Sou_obj),
    inherit_sig(M_name,Ar,_,_,Obj,_,Asou_obj),
    not(Asou_obj=Sou_obj).
sig_can_inherit(M_name,Ar,Vis,M_mode,Level,Obj,Sou_obj) :-
    inherit_sig(M_name,Ar,Vis,M_mode,Level,Obj,_,Sou_obj),
    not(conflict_sig(M_name,Ar,Obj)).
%% signature_inheritability_rules_end

%% method_inheritability_rules_start
meth_can_inherit0(M_name,Ar,M_mode,Obj,Obj) :- loc(M_name,Ar,Obj),
    sig_can_inherit(M_name,Ar,_,M_mode,ins,Obj,Sou_obj),
    tins(Obj,Sou_obj).
meth_can_inherit0(M_name,Ar,M_mode,Obj,Obj) :- loc(M_name,Ar,Obj),
    sig_can_inherit(M_name,Ar,_,M_mode,_,Obj,Sou_obj),
    tsub(Obj,Sou_obj).

```

Figure 3.1 XSB Binding of the Datalog++ Rulebase

```

meth_can_inherit0(M_name,Ar,M_mode, Obj,Sou_obj) :-
    loc(M_name,Ar,Sou_obj), tins(Obj,Sou_obj),
    ins(Obj,S_obj),
    meth_can_inherit0(M_name,Ar,M_mode,S_obj,Sou_obj),
    not(Obj=Sou_obj),
    not(loc(M_name,Ar,Obj)),
    sig_can_inherit(M_name,Ar,_,M_mode,ins,Obj,Sou1_obj),
    tsub(Sou_obj,Sou1_obj).
meth_can_inherit0(M_name,Ar,M_mode, Obj,Sou_obj) :-
    loc(M_name,Ar,Sou_obj), tsub(Obj,Sou_obj),
    sub(Obj,S_obj),
    meth_can_inherit0(M_name,Ar,M_mode,S_obj,Sou_obj),
    not(Obj=Sou_obj),
    not(loc(M_name,Ar,Obj)),
    not(rej(meth,M_name,Ar,Obj,S_obj)),
    sig_can_inherit(M_name,Ar,_,M_mode,_,Obj,Sou1_obj),
    tsub(Sou_obj,Sou1_obj).
conflict_meth(M_name,Ar,Obj) :-
    meth_can_inherit0(M_name,Ar,_,Obj,Sou_obj),
    meth_can_inherit0(M_name,Ar,_,Obj,Asou_obj),
    not(Asou_obj=Sou_obj).
meth_can_inherit(M_name,Ar,M_mode,Obj,Sou_obj) :-
    meth_can_inherit0(M_name,Ar,M_mode,Obj,Sou_obj),
    not(conflict_meth(M_name,Ar,Obj)).
%% method_inheritability_rules_end

%% method_visibility_rules_start
vis(_,_,Rec,Sen) :- tisa(Sen,Rec).
vis(_,_,Rec,Sen) :- Sen = Rec.
vis(M_name,Ar,Rec,Sen) :- class(Rec),
    sig_can_inherit(M_name,Ar,pub,_,_,Rec,_),
    not (Sen=Rec), not (tisa(Sen,Rec)).
vis(M_name,Ar,Rec,Sen) :- ins(Rec,S_obj),
    sig_can_inherit(M_name,Ar,pub,_,ins,S_obj,_),
    not (Sen=Rec), not (tisa(Sen,Rec)).
%% method_visibilty_rules_end

```

Figure 3.1 (continued)

programs. Appropriate actions are taken in translators/compiler to detect such an attempt and reject redefining of predicates in the rulebase.

Front-end System Design

The front-end system adopts a modular design. Figure 3.2 shows the overall system architecture.

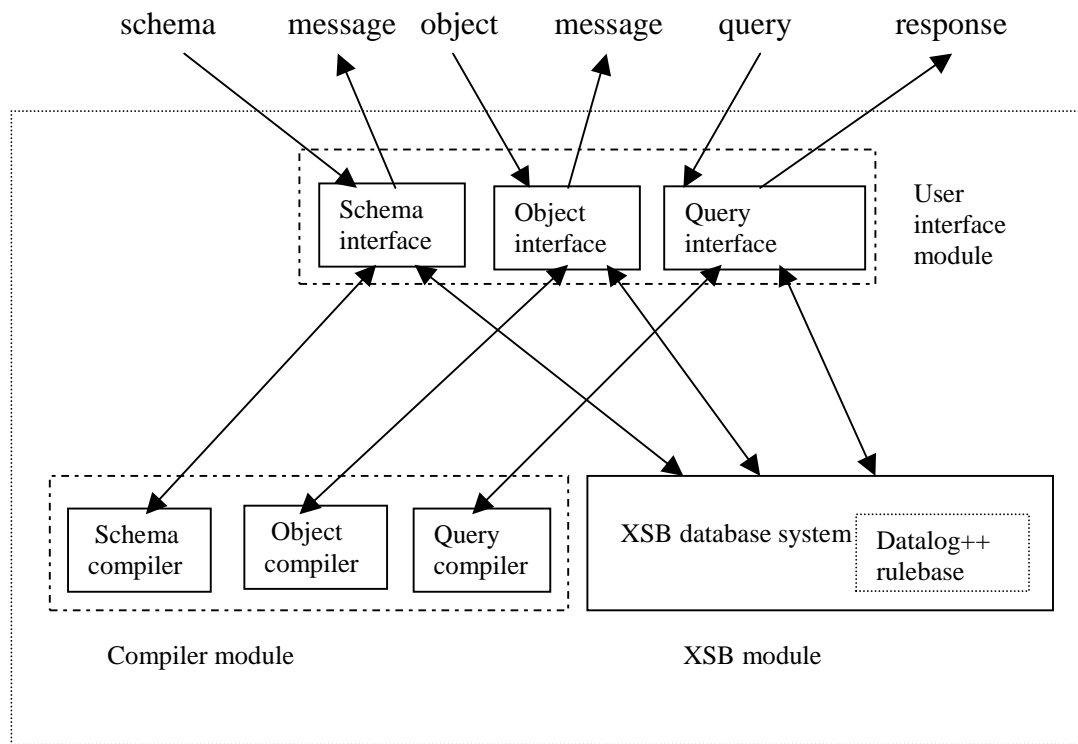


Figure 3.2. Front-end System Architecture

The system is composed of three modules: XSB, compiler, and user interface module. The user interface has a schema interface, an object interface, and a query interface to allow the user to deal with database operations such as schema definition,

object definition, and query definition, respectively. The three definitions correspond to the three components of the front-end Datalog++ syntax. Similarly, the compiler module includes a schema compiler, an object compiler, and a query compiler to translate schema definitions, object definitions, and query definitions, respectively.

Users interact with the system through the user interface module only. The user interface module also serves the control module of the whole system. It interacts with users, gets user requests, and dispatches tasks to the compiler module and the XSB module. The information flows as the following. The user's requests are directed to the user interface module in the forms of schema definitions, object definitions, or query definitions. The user interface module then passes schema definitions, object definitions, and query definitions to the compiler module and gets the translated versions and/or messages back. Messages are displayed to the user. Then the translated schema definitions, object definitions, and queries are passed on to the XSB module. Messages and/or answers are passed back to the user interface module and to the user.

Front-end System Implementation

The front-end system is implemented using Visual C++ and the Microsoft Foundation Class library on the Windows NT platform. Both the XSB and compiler modules are separately compiled and built into static libraries, which are linked to the user interface module to make a complete system. The rest of this section discusses the XSB, compiler, and user interface module.

XSB Module

XSB deductive database system comes with complete source code and instructions to build source code into a stand-alone application or a library. The decision to build XSB into a library was based on efficiency consideration. In this way, XSB becomes a component of the front-end system and function calls into the XSB module do not need to cross the process boundary.

One limitation of XSB (version 1.8) is that the XSB database cannot be re-initialized in a process's lifetime. The XSB deductive database system has not implemented necessary functions to clean up its memory space so that an old database can be unloaded and a new one can be loaded into the same space. Once a database is loaded into the memory space of the XSB deductive database system, the user cannot issue a command to forget the content of the database and then load a new database. Because of this limitation of the XSB database engine, the front-end currently does not support the operation of closing a database. Without cleanup support of the XSB, the current front-end is limited to one database during a process's lifetime. If re-initialization were supported in the XSB, the front-end could open and close databases freely. We hope that the next version of XSB will implement the re-initialization/cleanup functions.

Compiler Module

Language translation is a common task. Programmers can hand-code parsers and translators. They basically write recursive-descent parsers that recognize the input and do the translation. However, many compiler/translator construction tools exist to aid the task

of writing compilers/translators. In this project, PCCTS (Purdue Compiler Construction Tool Set) was used to build the schema, object, and query compilers. Using a compiler construction tool makes detection of grammar ambiguities and future enhancements of the language easier.

PCCTS (Parr 1996) is an integrated tool set, including a lexical analyzer generator and a parser generator. The parser generator accepts pred-LL(k) grammars (LL(k) grammars with syntactical and semantic predicate support). The user writes a grammar specification in the EBNF notation and embeds semantic actions for recognized grammar elements to do the actual translation. PCCTS accepts the grammar specification and generates a set of human-readable C++ source files to recognize and translate sentences in that language. The set of source files forms a top-down recursive-descent parser, similar to hand-coded recursive-descent parsers. Every grammar production rule has a corresponding function in the source files and this makes source-level debug much easier.

The compiler module translates the front-end Datalog++ syntax to the XSB syntax. During the translation, syntax and semantic errors are detected and reported by the compilers. Syntax errors are easy to catch because the grammar specifies the correct syntax. However, semantic errors have to be detected after a grammar element is correctly recognized by the parser. For example, the Datalog++ language dictates that a method can be defined only if its corresponding signature exists in the schema. To enforce this well-typedness rule, information of class definitions and class ISA hierarchies have to be collected, maintained, and consulted by the system when making a

decision. When the system see an object method definition, it checks if the corresponding signature is defined in this object or is inherited. If not, a semantic error is detected and reported to the user.

User Interface Module

The user interface module employs a set of user-friendly graphical interfaces to accept user input and present results and/or messages. The interfaces include an MDI editor, a schema dialog box, an object dialog box, a query dialog box, and a schema browser dialog box.

The MDI editor provides all standard editor functions such as copy, paste, etc. It also provides a status bar with line and column information of the current caret position in the active editor window. A "go to line" command is also provided to locate errors easily.

The schema and object dialog box handles compilation and loading of schema definition files and object definition files, respectively. The user should provide the file name of a definition file, click the "Check" button for compilation and the "Load" button for loading the compiled definitions into the XSB database. Messages (including errors, warnings, and status) are displayed in a message box. The user can switch from the schema/object dialog box to the MDI editor to locate the errors. After fixing the errors, the user can switch back to continue the compilation. Multiple schema/object definition files can be compiled and loaded as long as they do not provide conflicting information. However, unloading of a schema/object definition file from the XSB database is not

currently supported because this XSB version does not implement cleanup and re-initialization of the database.

The query dialog box handles compilation and submission of queries to the back-end XSB database engine. The user should provide the query and click the "Ask" button for processing. Messages (including errors, status, answers) are displayed in an answer box. There are two types of queries. One type is true/false queries that contain no variable in the query body. For this type of query, the front-end system answers "Yes" if the query is evaluated to be true by the XSB and "No" if the query is evaluated to be false. The other type asks the values of variables in the query body. For this type of query, the front-end system binds variables to values. Multiple answers may be returned. The set of answers is presented in a table format for easy interpretation. Each column represents a variable in the query and each row represents an answer.

The schema browser dialog box handles information browsing in the current database schema, including a set of classes with their superclasses, and their method signatures (local defined and inherited). The browser displays the set of classes in a list box. The user can select a class and click the "View" button to look at details for the class. For example, all available method signatures (either locally defined or inherited) are displayed. Since a method can only be implemented if an appropriate signature is available, displaying these signatures is very helpful to the user in writing object definitions.

CHAPTER 4

EVALUATION AND SUMMARY

This chapter evaluates the front-end system against the project contract and summarizes achievements of the project.

Evaluation

This goal of this project was to design and implement an object-oriented Datalog++ front-end for the XSB deductive database system. The resulting front-end software system meets the requirements specified in the project contract. The front-end system has been fully tested in the test plan. The following outlines how requirements in the project contract have been satisfied.

1. Requirement: The system should provide an editor for users to edit schema and object definitions.

Result: The front-end system has an integrated MDI editor, in which the user can open multiple windows to edit schema and object definitions.

2. Requirement: The system should parse schema and object definitions, and translate them and submit them to the XSB back-end in an interactive fashion.

Result: The front-end system has a schema dialog box and an object dialog box, which compile and load schema/object definitions, respectively.

3. Requirement: The system should include a schema analyzer to check well-typedness of object method definitions and to provide class information when requested by users.

Result: The front-end system does semantic checks, including well-typedness of object method definitions during translation. The front-end system also has a schema browser interface to view the database schema information.

4. Requirement: The system should provide a query interface for users to query the deductive database.

Result: The front-end system has a query interface for users to query the database.

5. Requirement: The system should display query responses in a user-friendly manner.

Result: The front-end displays query results in a table format.

Summary and Future Directions

Object-orientation and inference are considered two important features of the next generation database technology. Datalog++ is a deductive object-oriented database language and accounts for object-oriented features in a logical framework. This project provides a prototype implementation of the Datalog++ language over the XSB deductive database system.

The object-oriented Datalog++ front-end is a relatively complete database environment with schema, object, and query support. It utilizes services of the back-end XSB deductive database engine. The front-end system proves the feasibility of the

Datalog++ approach to integrating object-oriented features into deductive databases. It also enables users to take advantage of the object-oriented style of declarative database programming.

The front-end system can be enhanced in several ways. The current version of the front-end is intended as a prototype and supports only in-memory database. No disk persistence of the database is currently involved. Data persistence is an important service of a database system and should be supported in the next version. Secondly, during the lifetime of the front-end system startup, only one database is currently supported because XSB has not implemented the necessary cleanup and re-initialization functions. Future versions should allow users to freely close a database and open a new database.

REFERENCES

- Barja, Maria L., Alvaro A. A. Fernandes, Norman W. Paton, M. H. Williams, Andrew Dinn, and Alia. I. Abdelmoty. 1995. Design and implementation of ROCK & ROLL: A deductive object-oriented database system. *Information Systems* 20 (3): 185-211.
- Chen, W., and D. S. Warren. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM* 43 (1): 20-74.
- Frohn, J., R. Himmeroder, P. T. Kandzia, and C. Schleppehorst. 1997. FLORID User Manual, Version 2.0, November 1997. <http://www.informatik.uni-freiburg.de/~dbis/florid/help.html> (Accessed 23 February, 1999)
- Jamil, Hasan M., and Laks V. S. Lakshmanan. 1992. ORLOG: A logic for semantic object-oriented models. In *Proceedings of the ISMM first international conference on information and knowledge management held in Baltimore, Maryland, November, 1992*, 584-92.
- Jamil, Hasan M. 1997. Implementing abstract objects with inheritance in Datalog^{neg}. In *Proceedings of the 23rd VLDB conference held in Athens, Greece, 1997*.
- Kifer, M., and G. Lausen. 1989. F-Logic: A higher-order language for reasoning about objects, inheritance, and schemes. In *Proceedings of the ACM SIGMOD conference on management of data held in Portland*, 134-146.
- Liu, Mengchi. 1996. ROL: A deductive object base language. *Information Systems* 21(5): 431-57.
- Parr T. J. 1996. *Language translation using PCCTS & C++: A reference guide*. San Jose, CA: Automata Publishing Company.
- Sagonas, Konstatinos F., Terrance Swift, David S. Warren, Juliana Freire, and Prasad Rao. 1998. *The XSB programmer's manual, version 1.8*. <http://www.cs.sunysb.edu/~sbprolog/manual/manual.html> (Accessed 8 February, 1999).

- Srivastava, Divesh, Raghu Ramakrishnan, Praveen Seshadri, and S. Sudarshan. 1993. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the international conference on very large data bases held in Dublin, Ireland*, 158-70.
- Zaniolo, Carlo, Stepano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roberto Zicari. 1997. *Advanced database systems*. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

APPENDIX A
PROJECT CONTRACT

PROJECT CONTRACT

STUDENT: Zhixin Tang

SSN: 206-74-6735

PROJECT TITLE

Design and Implementation of the Datalog++ Front-end for XSB-PC

PROJECT DESCRIPTION

Datalog++ is a deductive object-oriented database language proposed by Dr. Hasan Jamil. It incorporates most of the salient object-oriented features, such as encapsulation with and inheritance with overriding and conflict resolution, into the Datalog language. In Datalog++, users can write classes and objects to model their application domains. The system then translates the program written in Datalog++ into a language that a back-end deductive database system such as XSB understands.

This project is to implement a Datalog++ front-end for XSB on the PC platform, which is a public-domain deductive database system developed at SUNY-Stony Brook. The front-end is intended to be an abstraction layer over XSB, rather than an extension of XSB. So, it will only support features specific to the Datalog++.

The front-end will be implemented on Windows NT platform using Visual C++. It will provide graphical user interfaces for users to edit programs and to run programs. There will be a schema interface, an object interface, and a query interface. The exact design of the interface will be decided based on the HCI principles.

OBJECTIVES

The main objective of the project is to design and implement an objected-oriented front-end to an existing deductive database (XSB) so users can take advantage of the object-oriented style of declarative database programming.

SYSTEM FUNCTIONS

1. The system should provide an editor for users to edit schema and object definitions.
2. The system should parse schema and object definitions, and translate them and submit them to the XSB back-end in an interactive fashion.
3. The system should include a schema analyzer to check well-typedness of object method definition and to provide class information when requested by users.
4. The system should provide a query interface for users to query the deductive database.
5. The system should display query responses in an user friendly manner.

CONSTRAINTS AND CONSIDERATIONS

The front end only supports essential features of deductive database systems. It is intended as a prototype and will support in-memory database only. No disk persistence of database will be involved. However, stored programs may be consulted, modified, added to in-memory programs, saved, and executed. The system should provide an incremental compilation and program development environment.

PROJECT DELIVERABLES

The deliverables in this project are:

1. Datalog++ language interface
2. Design document
3. Source code that performs the specified function above
4. Test plan and testing
5. User manual
6. Project report

Approved:

Dr. Hasan Jamil
Assistant Professor of Computer Science
(Major Professor)

Dr. Susan Bridges
Associate Professor of Computer Science
(Committee Member)

Dr. Nancy Miller
Associate Professor of Computer Science
(Committee Member)

APPENDIX B

LIST OF PROJECT DELIVERABLES

LIST OF PROJECT DELIVERBLES

- Tang, Zhixin. Datalog++ Front-end for XSB: Datalog++ Language Specification. Version 1.0. Department of Computer Science, Mississippi State University, March 1999.
- Tang, Zhixin. Datalog++ Front-end for XSB: Software Design Description. Version 1.0. Department of Computer Science, Mississippi State University, March 1999.
- Tang, Zhixin. Datalog++ Front-end for XSB: Test plan. Version 1.0. Department of Computer Science, Mississippi State University, March 1999.
- Tang, Zhixin. Datalog++ Front-end for XSB: User Manual. Version 1.0. Department of Computer Science, Mississippi State University, March 1999.
- Tang, Zhixin. Datalog++ Front-end for XSB: Source Code. Version 1.0. Department of Computer Science, Mississippi State University, March 1999.

APPENDIX C

SOME SNAPSHOTS OF THE FRONT-END

SOME SNAPSHOTS OF THE FRONT-END

In this appendix, we show several snapshots to give a flavor of the front-end interfaces. For detailed information, please consult with the technical documents user's manual and test plan. Figures A3-1, A3-2, A3-3, A3-4, and A3-5 are snapshots of the Datalog++ front-end start-up window, schema loader interface, object loader interface, schema browser interface, and query interface, respectively.

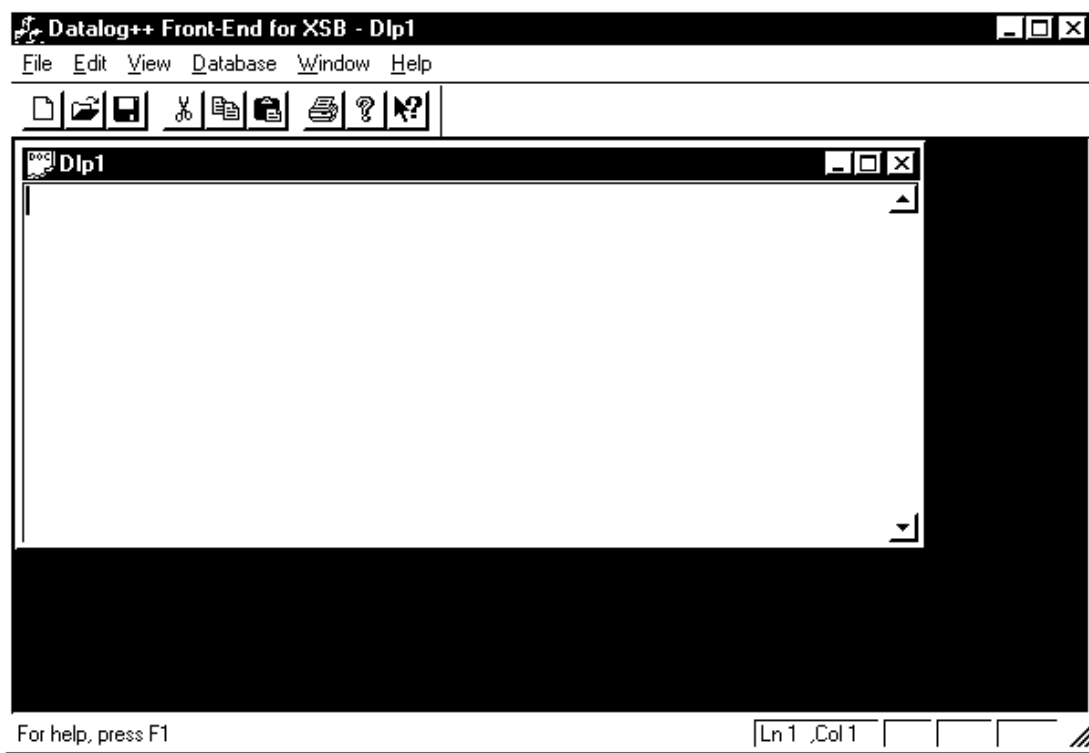
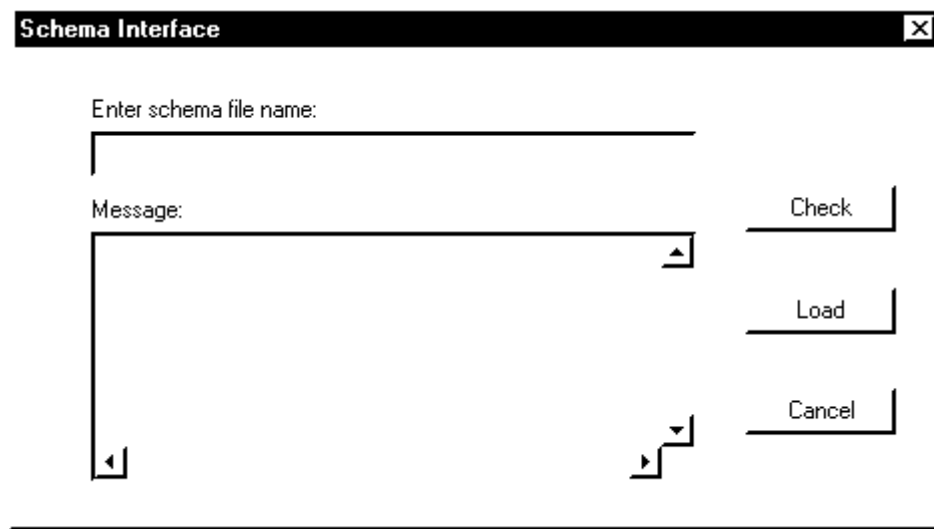
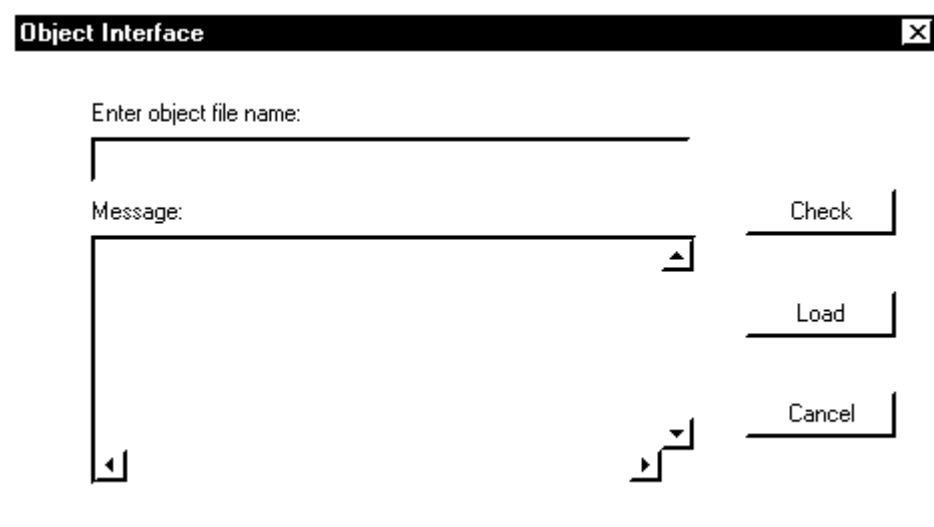


Figure A3-1. Start-up Window



The screenshot shows a window titled "Schema Interface" with a close button in the top right corner. Inside the window, there is a text input field labeled "Enter schema file name:". Below this is a message box labeled "Message:" with a scrollable area. To the right of the message box are three buttons: "Check", "Load", and "Cancel".

Figure A3-2. Schema Loader Interface



The screenshot shows a window titled "Object Interface" with a close button in the top right corner. Inside the window, there is a text input field labeled "Enter object file name:". Below this is a message box labeled "Message:" with a scrollable area. To the right of the message box are three buttons: "Check", "Load", and "Cancel".

Figure A3-3. Object Loader Interface

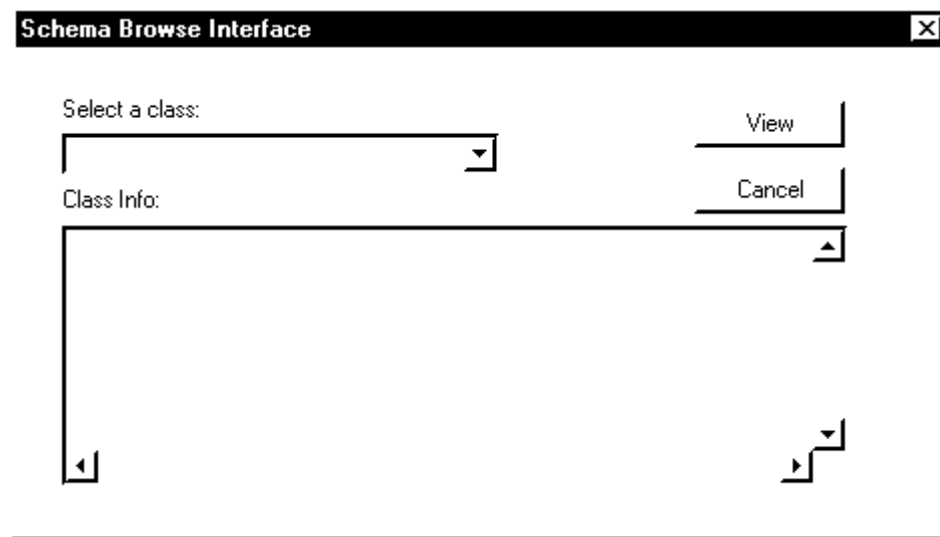


Figure A3-4. Schema Browser Interface

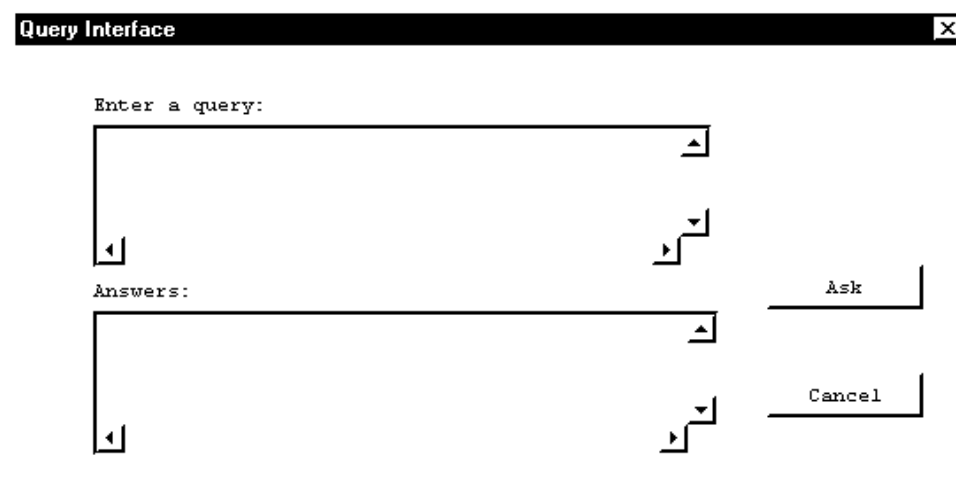


Figure A3-5. Query Interface

APPENDIX D
SOME EXAMPLE TEST CASES

SOME EXAMPLE TEST CASES

In this appendix, we show several example test cases. All the source code files referred in this Appendix can be found under the subdirectory “examples\test\” once the system is installed. Complete tests are done in the test plan document. Please refer to that document for details.

Example 1: Detect Semantic Errors in a Schema Definition File

The schema file “error.s” has several semantic errors in it. Comments in the file point out these errors. The contents of the file is listed below:

```
// File name: error.s
// Description:
// This is a schema file with the following semantic errors that the system catches:
// ERROR: duplicate signature
// ERROR: duplicate class
// ERROR: superclass not defined
//ERROR: reject from non-immediate superclass

CLASS c0
{
    instance signatures
    {
        pub,val m1/1;
        pub,code m2/1;
        priv,code m1/1; //ERROR: duplicate signature
    }
}

// ERROR: duplicate class
CLASS c0
{
    class signatures
    {
        pub,val m2/1;
    }
}
```



```

CLASS c1 subclass of {c0,c2} // ERROR: superclass c2 not yet defined
{
}

CLASS c2 subclass of {c0}
{
  instance signatures
  {
    pub,val m2/1;
  }
}

CLASS c3 subclass of {c1,c2}
{
  controls
  {
    reject sig m2/1 from c2;
    reject meth m1/1 from c0; //ERROR: only reject from immediate superclass
  }
}

```

Open the schema loader dialog, check the schema "error.s". The following is the actual output:

```

*****Begin parsing class*****
ERROR:
Duplicate signature declaration: m1/1
At or before line 17

ERROR:
Duplicate class declaration: c0
At or before line 22

ERROR:
Superclass not defined: c2
At or before line 30

ERROR:
Reject from non-immediate superclass: c0
At or before line 49

WARNING: Multiple inheritance signature conflict in c1
signature: m2/1 will not be inherited

```

WARNING: Multiple inheritance signature conflict in c1
signature: m2/1 will not be inherited

***** Parse class failed. *****

The errors are all detected by the front-end system. This test is successful.

Example 2: Detect Semantic Errors in an Object Definition File

In order to test semantic errors in object definition files, a correct schema has to be in the database system. First load a correct schema definition file “correct.s”, then go to the object loader interface and check the object file “error.o”. The contents of files “correct.s” and “error.o” are listed below. Comments in the file “error.o” shows the errors to be detected.

```
// File name: correct.s
// Description:
// This schema file is a correct version of error.s
// Load this schema into database before test error.o
```

```
CLASS c0
{
    instance signatures
    {
        pub,val m1/1;
        pub,code m2/1;
    }
}
```

```
CLASS c1 subclass of {c0}
{
}
```

```
CLASS c2 subclass of {c0}
{
    instance signatures
    {
        pub,val m2/1;
    }
}
```

```

}

CLASS c3 subclass of {c1,c2}
{
    controls
    {
        reject sig m2/1 from c2;
    }
}

// File name: error.o
// Description:
// This is an object definition file with the following semantic errors that the system catches:
// ERROR: attempt to define method whose signature not available
// ERROR: duplicate object(instance) name
// ERROR: attempt to redefined system-reserved predicates(only test predicate class/1 here)

// The list of system-reserved predicates:
// class, object, ins, tins, tsub, tisa, sig, rej, loc, meth, vis, meth_can_inherit,
// sig_can_inherit, inherit_sig, conflict_sig, meth_can_inherit0, conflict_meth, bagCount,
// bagMin, bagMax, bagSum, bagAvg, minimum, maximum, successor, sum, sum_count

// LOAD schema file correct.s before this test

c1.m1(1);
c1.m2(X) :- m1(X);

c1.m3(1000); // ERROR: method m3 has no signature (inherited or local)

c2.m2(2);

o3:c3;
o3.m1(3);

o3:c1; // ERROR: duplicate object(instance) name o3

class(c1000); // ERROR: try to redefined system-reserved predicate class
meth_can_inherit(m3,1,c3, c2); // ERROR: try to redefined system-reserved predicate meth_can_inherit

```

Load a correct schema file "correct.s", then check object definition file "error.o".
The following is the actual output:

```

*****Begin parsing object*****
ERROR:
No appropriate signature exists: c1.m3/1
At or before line 20

ERROR:
Duplicate object declaration: o3

```

At or before line 27

ERROR:

Cannot redefine system predicate: class

At or before line 29

ERROR:

Cannot redefine system predicate: meth_can_inherit

At or before line 30

***** Parse object failed. *****

All errors are detected by the system. This test is passed.

Example 3: Test Datalog++ Language Features by Queries

Load the schema definition file “test.s” and object definition file “test.o” into the database. The contents of files “test.s” and “test.o” are listed below. See the comments in the file “test.o” for the set of queries tested and the expected results.

```
// File name: test.s
// Description:
// Load test.s and test.o to test Datalog++ features
```

```
CLASS c0
{
  class signatures
  {
    pub,val m0/1;
  }

  instance signatures
  {
    pub,val m1/1;
    pub,code m2/1;
    pub,val m3/1;

    priv,val m4/1;
    pub,val m5/1;
  }
}
CLASS c1 subclass of {c0}
```

```
{
}
```

```
CLASS c2 subclass of {c0}
{
    instance signatures
    {
        pub,val m2/1;
    }
}
```

```
CLASS c3 subclass of {c1,c2}
{
}
```

```
CLASS c4 subclass of {c1,c2}
{
    controls
    {
        reject sig m2/1 from c2;
    }
}
```

```
CLASS c5 subclass of {c1,c0}
{
}
```

```
// File name: test.o
// Description:
// Load test.s and test.o to test Datalog++ features:
// 1. inheritance ambiguity, reject all
//   c3<<m2(X); // ambiguity, answer: empty
//   c5<<m2(X); // no ambiguity, common source, answer: 1
// 2. user-specified ambiguity resolution: reject
//   c4<<m2(X); // rejected one inheritance, answer: 1
// 3. class sig/meth cannot be inherited to instances
//   o4<<m0(X); // m0 is a class sig/meth, answer: empty
//   c4<<m0(X); // answer: 0
// 4. code/val inheritance
//   o4<<m1(X); // code "m2(X) :- m1(X)" inherited, answer: 3
//   o4<<m3(X); // value 1 inherited, answer: 1
// 5. access control: pub/priv
//   c0<<m4(X); // private, answer: empty
//   c0<<m5(X); // in the class/subclass, can access private method
//       answer: 20
//
```

```
c0.m0(0);
c0.m4(10); // private method
c0.m5(Y) :- m4(X), Y is 2*X;
```

```
c1.m1(1);
c1.m2(X) :- m1(X);
c1.m3(X) :- m1(X);
```

```
c2.m2(2);
```

```
o4:c4;
o4.m1(3);
```

Conceptually, the database can be viewed as in the figure on the next page. If the following queries are asked, the actual answers are shown as comments after //. Actual answers match the expected answers. This test is passed.

1. inheritance ambiguity, reject all
 - c3<<m2(X); // ambiguity, answer: empty
 - c5<<m2(X); // no ambiguity, common source, answer: 1
2. user-specified ambiguity resolution: reject
 - c4<<m2(X); // rejected one inheritance, answer: 1
3. class sig/meth cannot be inherited to instances
 - o4<<m0(X); // m0 is a class sig/meth, answer: empty
 - c4<<m0(X); // answer: 0
4. code/val inheritance
 - o4<<m1(X); // code "m2(X) :- m1(X)" inherited, answer: 3
 - o4<<m3(X); // value 1 inherited, answer: 1
5. access control: pub/priv
 - c0<<m4(X); // private, answer: empty
 - c0<<m5(X); // in the class/subclass, can access private method
answer: 20

