Implementation of a Graph Oriented Query Language : IUGQL

Vijay M. Sarathy Dirk Van Gucht

Indiana University
Computer Science Department
Bloomington, Indiana 47405-4101, USA
e-mail: {vijay,vgucht}@cs.indiana.edu

Abstract

In this paper, we describe the implementation of IUGQL[1], a graph oriented query language for object oriented databases that is based on the Graph Oriented Object Data (GOOD) model. In the implementation, databases (graphs) are represented as a set of (named) binary relations. The graph oriented queries are translated into equivalent Tarski algebra expressions. These Tarski expressions manipulate the binary relations to yield the desired query result.

[1]IUGQL is an acronym for **I**ndiana **U**niversity **G**raphical **Q**uery **L**anguage.

1 Introduction

Relational database systems have succeeded commercially because of their simplicity and strong theoretical foundation. However, in recent years, driven by the need for database applications involving complex objects, new concepts have arisen suggesting further approaches to database systems. These new concepts have combined ideas from several areas of computer science, most notably the object oriented paradigm, into database technologies.

To incorporate these concepts into databases, a variety of new database models have been introduced. These database models can be classified into three main categories: the *complex object* models, the *function-based object* models, and *hybrids* of these. Complex object models extend the relational model by allowing, besides flat relations, complex object types such as tuples, sets, lists, arrays, etc. Function-based object models, on the other hand, view a database as a graph of objects organized in classes, where the links between objects express single-valued and multi-valued functions (relationships) between objects. Although researchers have succeeded in extending the well-known theory of relational query languages to complex object models, the study of query languages for function-based object databases is much less developed.

Given this situation, we considered a *simple* algebra, the *Tarski algebra*, that is appropriate to support function-based object query languages [7, 12, 13]. Unlike previously considered algebras, the Tarski algebra operates on graphs (interpreted as conceptual binary relations) rather than on objects of complex types. In that respect, the Tarski algebra is at the level of abstraction of function-based object models and is thus more natural and effective than other algebras for such database models. In [12], we showed how to translate queries specified in a graph-oriented query language into the Tarski algebra. A graph-oriented query specification language was chosen because graphs are the natural representation of function-based object databases [6, 10].

In this paper, we describe the implementation of IUGQL, a graph oriented query language for object oriented databases, using the Tarski algebra as an underlying algebraic foundation. In addition to the implementation of the graphical query language, we have also successfully implemented optimization techniques proposed in [12] to provide for efficient query processing.

In Section 2 we provide a brief overview of the graph oriented model, graph oriented queries, the Tarski algebra, and the translation algorithm to effectively translate queries in the graph oriented query language into the Tarski algebra. The reader is referred to [6, 12, 13] for more details. In Section 3, we describe the overall layout of the implementation of IUGQL. In Section 4, we discuss the lower level components of IUGQL, the Tarski algebra interpreter and the Tarski machine. In section 5 we discuss the higher level components of IUGQL, the graph-tarski optimizer and translator. We discuss the graph primitives used to implement the graph oriented query language and the implementation of the translation algorithms and graph optimization techniques. In Section 6 we provide details on how to use the system. In Section 7 we provide detailed example queries and the output produced by IUGQL, which can be used a tutorial introduction on how to use IUGQL. In Section 8 we provide directions for future work and research. We provide a few appendices to highlight the important code header fragments and other relevant details.

2 The Graph Oriented Query Language, the Tarski Algebra, and the Translation Algorithms

2.1 The Graph Oriented Query Language

The graph-oriented representation of a function-based object database views the database as a labeled directed graph. The labeled nodes correspond to *objects* in the database, and

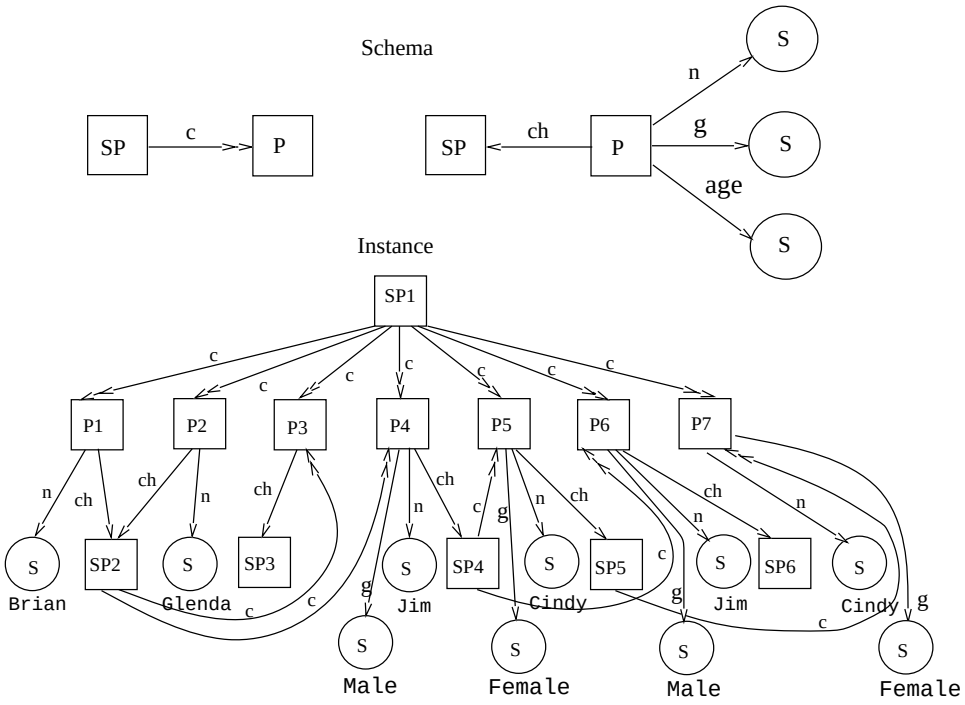


Figure 1: An object base schema of a persons database and an instance over the persons schema.

the labeled edges correspond to *functions* or relationships between the objects [6].

Consider the graph in Figure 1 (top). It represents the *object base schema* of a persons database. The rectangular nodes represent *structured object classes* that can have a complex internal structure, whereas the circular nodes represent *basic object classes*. The edges in the schema denote *functions*, i.e., *properties* or relationships between object classes. There are two possible function types, *single-valued* ($\rightarrow$) function types, and *multivalued* ($\rightarrow\!\!\rightarrow$) function types[2].

The graph shown in Figure 1 (bottom) is an example of an *object base instance* over this persons schema. Also attached to each basic object is its value. Notice how each object, with its properties, agrees with the schema.

Given this graph representation of a database, it is natural to specify queries in the form

[2]The function (property) (SP, c, P) is multi-valued since a set of persons can consist of *several* persons.

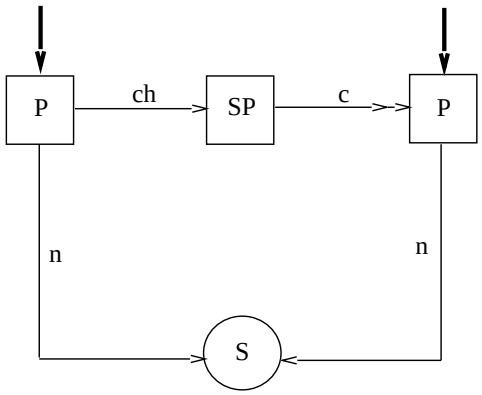


Figure 2: A query graph over the persons object base specifying named parent-child pairs who have the same name. The two P nodes are the *selected* nodes.

of *query graphs.* A *query graph*, relative to a given object base instance, defines a set of *embeddings* which are the subgraphs of the object base instance that match the query graph.

Consider the following query: Find all parent-child pairs who have the same name. In Figure 2, we show a query graph that represents the above query[3]. Since we are interested in the two P nodes as output, we designate them as *selected nodes* (pointed to by bold arrows) in the query graph as shown in Figure 2.

In addition to the query graph, the graph oriented model also specifies *actions* to perform transformation operations on the database. The model allows specification of actions like *node addition*, *edge addition*, *node deletion*, and *edge deletion* [6]. Our implementation implements all of the actions specified in the model.

2.2 The Tarski Algebra

Since the object base schemas and instances we are working with are labeled graphs, a natural (and easy) way to conceptualize them is as a collection of binary relations. To manipulate these conceptual relations and adhere to the function-based approach, it is necessary to

[3]For the instance in Figure 1, this query graph defines two embeddings corresponding to the two parent-child pairs, {P4,P6} and {P5,P7}.

develop an algebra that is closed with respect to the class of binary relations and that is expressive enough to handle all reasonable queries.

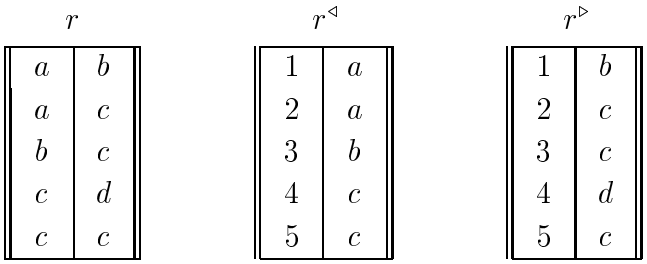
In the 1940's Alfred Tarski proposed an algebra to manipulate binary relations [16]. The kernel of the basic Tarski algebra consists of four well-known operators on binary relations: union ($r \cup s$), relation composition ($r \odot s$), inverse (r^{-1}), and (finite) complementation ($\overline{r}$)[4]. This algebra was extended in [7, 12] to enable representation of complex objects by adding certain *object-id creation* operators. The algebra was also extended with some simple *selection* operators. The selection operators are very simple and select pairs from a relation based on certain selection conditions involving constants. The object-id creation operators are more fundamental and allow the creation of object identifiers for ordered-pairs. Specifically, the full Tarski algebra has, besides the four basic operators, two *constant selection* (the *left-* and *right selection*) operators, and two *ordered-pair oid creation* (the *left-* and *right oid creation*) operators. This grammar was further extended in [14] to facilitate parallel execution of queries by allowing mechanisms to horizontally partition relations and accumulate subquery results.

In Figure 3, we show the result of *left* and *right-oid creation* on a relation r, denoted $r^{\triangleleft}$ and $r^{\triangleright}$, respectively. Each ordered pair in r has received a separate oid. The left-value (right-value) of that pair is found in $r^{\triangleleft}$ ($r^{\triangleright}$).

The oids that our system generates are based on either one of two strategies, the *tuple sensitive* strategy, or the *relation sensitive* strategy. In tuple sensitive oid creation, the relation name is ignored, and a concatenation of the left and right values of the pair is used as the oid. In relation sensitive oid creation, the relation name is prefixed to the concatenation of the values of the pair to generate the oid.

There are also some *derived* Tarski operators that are useful in the translation process.

[4]For a detailed explanation of the operators, see [12, 13].



r	
a	b
a	c
b	c
c	d
c	c

$r^{\triangleleft}$	
1	a
2	a
3	b
4	c
5	c

$r^{\triangleright}$	
1	b
2	c
3	c
4	d
5	c

Figure 3: Example of left and right oid creation on a relation r.

They are r^{π}, r^{π_l}, r^{π_r}, r^{τ}, and $r \cap s$, and are defined in [12].

2.3 Encoding an Object Base as Conceptual Binary Relations

Reconsider the (labeled-directed) graph in Figure 1 which represents the schema and an instance of the persons object base. This object base can be encoded as a set of binary relations, by storing each edge type as a binary relation, and inserting the left and right nodes of the edge in the graph as pairs in the corresponding relation. Nodes are stored as relations with identical pairs.

The conceptual binary relation instance or the *Tarski instance* for the persons object base is shown in Figure 4.

2.4 The Translation Algorithm

The translation algorithm is a graph reduction algorithm which uses the techniques of *edge reduction*, *node combination*, *projection*, *chaining*, *disconnected node combination*, and *selection* [13].

The key problem in the query graph translation process is to derive an algebraic expression for the search conditions specified in the query graph of Figure 2. We briefly discuss

Tarski Instance corresponding to Structured Objects

SP

SP_l	SP_r
SP1	SP1
SP2	SP2
SP3	SP3
SP4	SP4
SP5	SP5
SP6	SP6

P

P_l	P_r
P1	P1
P2	P2
P3	P3
P4	P4
P5	P5
P6	P6
P7	P7

Tarski Instance corresponding to Properties

c

SP	P
SP1	P1
SP1	P2
SP1	P3
SP1	P4
SP1	P5
SP1	P6
SP1	P7
SP2	P3
SP2	P4
SP4	P5
SP4	P6
SP5	P7

ch

P	SP
P1	SP2
P2	SP2
P3	SP3
P4	SP4
P5	SP5
P6	SP6

n

P	n
P1	Brian
P2	Glenda
P4	Jim
P5	Cindy
P6	Jim
P7	Cindy

g

P	g
P4	Male
P5	Female
P6	Male
P7	Female

Figure 4: The conceptual Tarski instance of the persons object base.

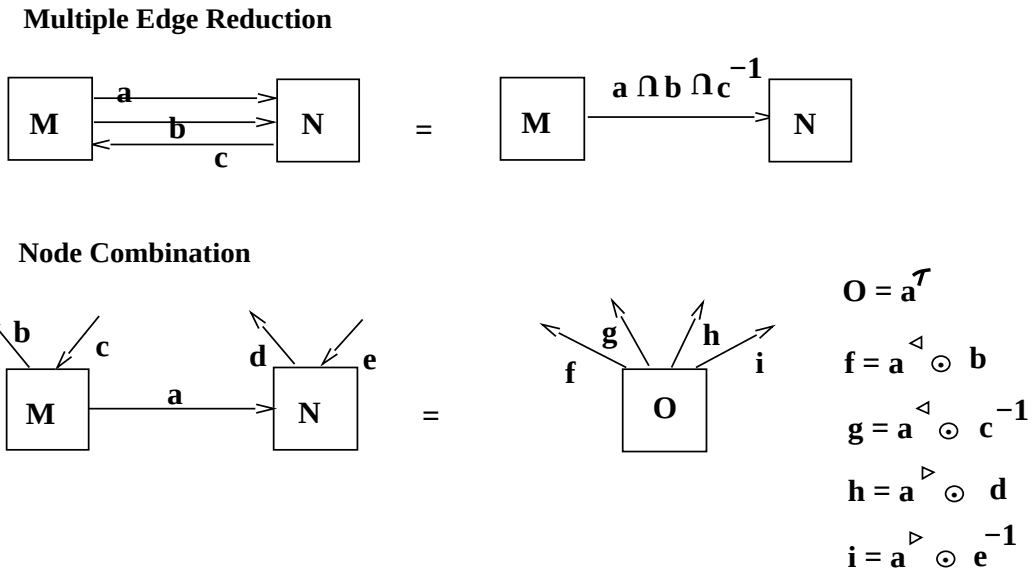


Figure 5: The main techniques used in the query graph translation algorithm.

the techniques of *edge reduction* and *node combination* in this section[5] as they are the most important in the translation process.

1. The *multiple edge reduction* technique.

   In this technique, multiple function edges between a pair of objects in the query graph are replaced by a single function edge that represents the intersection of all the edges. This reduces the query graph by at least one edge.

2. The *node combination* technique.

   In this technique, two objects connected by a function edge in the query graph are combined into one composite object (using oid-creation), and the edges leaving/entering the two nodes are simulated by appropriate compositions[6]. This reduces the query graph by at least one edge and one node.

The techniques of edge reduction and node combination are summarized in Figure 5. By repeatedly applying these graph reduction techniques, it is possible to reduce the query graph to a single node. This node (and its associated Tarski algebra expression) encodes the algebraic solution to the query graph.

Once we have applied the translation algorithm to the query graph in Figure 2, we have to compute the result of the query. This is accomplished by backtracking the sequence of node combinations performed during the translation process and computing the projections onto the selected nodes [12, 13]. The actions are translated by appropriately creating new binary relations and/or side-effecting existing binary relations [12, 13].

[5]The implementation of all the techniques used is discussed in more detail Section 5.3.

[6]For more details, see [12, 13].

3 Overall Layout of the Implementation of IUGQL

As indicated in Figure 6, the implementation of IUGQL has two major components, the lower level query processing component, and the higher level graph translator component.

At the lower level of the implementation, we implement binary relations on top of the EXODUS storage manager [3] using the E programming language [4]. Facilities for creating, loading, and storing binary relations are provided. These binary relations constitute the underlying storage structure of the system. On top of this storage structure, we have implemented an interpreter for the Tarski algebra and a Tarski machine to facilitate the manipulation of the binary relations using Tarski algebra expressions. The Tarski interpreter and machine have been written in C using yacc to parse Tarski algebra expressions. The interpreter and machine also support the extended Tarski algebra operators outlined in [14] that facilitate specification of parallel execution of queries.

At the higher level of the implementation, we provide a Graph-Tarski translator and optimizer that translate graph oriented queries into equivalent optimized Tarski algebra expressions. The relevant graph primitives required to create and manipulate graphs were identified and implemented in C++. The query graph translation algorithms were then implemented using the graph primitives. This component of the implementation takes a graph[7] as input and returns an equivalent Tarski algebra expression as output which is then evaluated by the lower level Tarski interpreter and machine to provide the final result of the query.

We have completed a working prototype of IUGQL and are now planning on adding an X-window based graphical front-end to the system to enable users to input graphs via the interface.

[7] In this version of IUGQL, graphs are input textually.

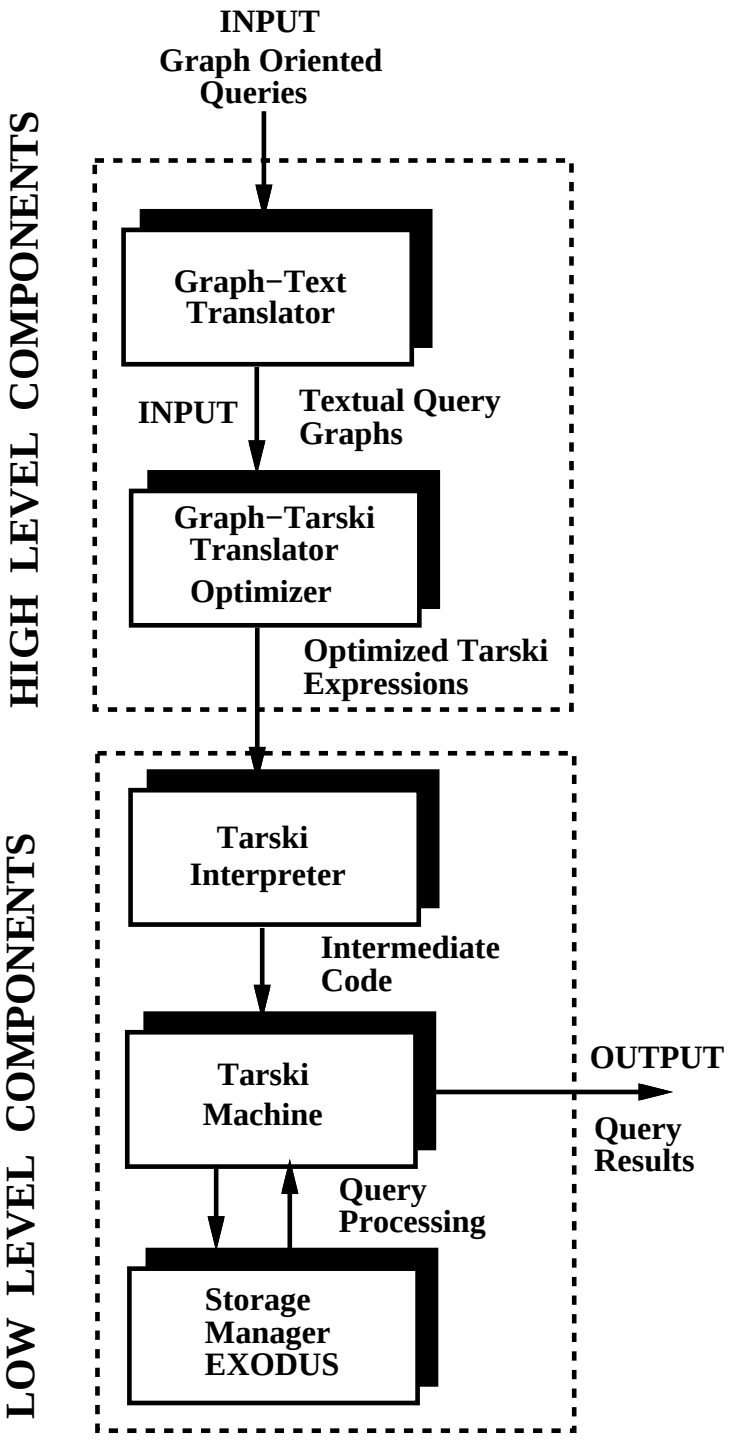


Figure 6: The Layout of the Implementation of IUGQL.

4 Lower Level Components of IUGQL

4.1 The Tarski Interpreter

The interpreter for the extended Tarski algebra was implemented in C using yacc to parse Tarski algebra expressions. The grammar (specified in yacc) for the interpreter is shown below.

```
/* Grammar for the Tarski algebra */

expr:    /* nothing */
          |  expr '\n'
          |  expr bexpr '\n'       /* create/load/append/delete expressions */
          |  expr aexpr '\n'       /* assignment expressions */
          |  expr rexpr '\n'       /* basic tarski expressions */
          |  expr eexpr '\n'       /* extended tarski expressions */
          |  expr error '\n'       /* error detection */
          ;
bexpr:      CREATE '(' REL ')'     /* create new binary relation */
          | APPEND '(' rel ')'     /* append to existing relation */
          | LOAD '(' REL ')'       /* load existing relation from EXODUS */
          | PREOPRTR '(' rel ')'   /* PREOPRTR can be append, delete,
                                      scan or store */
          | DELETEVAR '(' VAR ')'
          | NEW '(' ')'
          | SHOW_DICT              /* SHOW_DICT lists all the relations
                                      and variables in the symbol table */
          | REL_OID                /* change the oid-creation strategy to
                                      relation sensitive */
          | TUP_OID                /* change the oid-creation strategy to
                                      tuple sensitive */
          | OID_STRAT              /* show current oid-creation strategy */
          ;
rel:        REL                    /* relation name */
          ;
rexpr:      OPRTR '(' rexpr ')'    /* (unary) OPRTR can be id, lid, rid, tau,
                                      inv, compl, compl1, ltag, rtag, stag,
                                      sstag, new, or closure */
          | OPRTR '(' rexpr ',' rexpr ')' /* (binary) OPRTR can be compose,
```

```
                                        necompose, lecompose, gecompose,
                                        union, diff, inter, or cross */
          | IF '(' rexpr ')'  THEN '(' rexpr ')' ELSE '(' rexpr ')'
          | NIF '(' rexpr ')'  THEN '(' rexpr ')'  ELSE '(' rexpr ')'
          | rel
          | REL '=' rexpr                 /* assignment */
          | SELR '(' CNST ',' rexpr ')'   /* right selection with constant */
          | SELR '(' VAR  ',' rexpr ')'   /* right selection with variable */
          | SELL '(' CNST ',' rexpr ')'   /* left selection with constant */
          | SELL '(' VAR  ',' rexpr ')'   /* left selection with variable */
          | '(' rexpr ')'
          | '(' VAR ',' VAR ')'           /* used for enumeration */
          | '{' rexpr ',' rexpr '}'       /* which is useful      */
          | '{' rexpr ',' aexpr '}'       /* in                   */
          | '{' aexpr ',' rexpr '}'       /* the                  */
          | '{' aexpr ',' aexpr '}'       /* extended Tarski algebra */
          | '{' '}'                       /* empty relation */

          |  eexpr                        /* extended Tarski expression */
        ;
aexpr:      VAR '=' CNST                  /* constants should be enclosed in
                                             double quotes, eg. "John" */
          | VAR                           /* variable names should be
                                             preceded and followed by
                                             _ (underscore), eg. _x_  */
        ;
eexpr:      for '(' eexpr ')' eop '(' eexpr ')' end  /* for expression */
          | rexpr
        ;
for:        FOR '[' VAR ',' VAR ']' IN
        ;
eop:        EOPRTR                       /* EOPRTR can be eunion, einter,
                                              or ediff */
        ;
end:       /* nothing */
```

4.2 The Tarski Machine

The Tarski machine is implemented as a simple *stack machine*. The *actions* taken by the Tarski interpreter for every *rule* in the grammar, generate intermediate code instructions that get pushed onto the *execution stack*, which is an array of pointers. The pointers on the execution stack point to either built-in Tarski operator functions or to data in the symbol table. The machine is generated during parsing by pushing the appropriate instructions onto the execution stack. As and when subexpressions are evaluated depending on the instructions on the execution stack, they get pushed onto an *interpreter stack*, which contains either the contents of a relation or variable, or a pointer to an appropriate symbol table entry.

Each cycle of the machine executes the instruction (function) pointed to by the program counter, and increments the counter, so it is ready for the next instruction. Of course, the program counter may also be incremented to step over arguments to instructions and point to further instructions. The execution of the machine is fairly straightforward, as instructions on the execution stack are executed and intermediate results pushed and popped from the interpreter stack. The only complicated case is the case of the execution of the *for* expression of an extended Tarski expression. This is accomplished by recursive calls to the execution of the machine for each subexpression in the extended Tarski expression.

Binary relations can be created interactively, and then stored persistently using the EXODUS storage manager. Persistent storage facilities for storing/loading binary relations are provided using the E programming language. The functions that implement the built-in unary and binary Tarski operators are loaded into a symbol table when the program is invoked. Relations and variables created during the execution of the interpreter are also stored in the symbol table for efficient access. Binary relations in the symbol table are stored as a linked list of attribute pairs. Temporary relations evaluated during the computation of a Tarski expression are stored in a different symbol table to enable common sub-expression

optimization.

4.2.1 EXODUS Storage Structure for Binary Relations

Binary relations are stored persistently using the EXODUS storage manager. Each pair in a binary relation is declared as a class with a left and right attribute in the E programming language as shown below.

```
dbclass rel_persist {
 public:
  dbchar left[1000];
  dbchar right[1000];
};
```

A complete binary relation is stored as a *collection*[8] of such attribute pairs. The database (set of all binary relations) is stored as a *persistent array* of such *collections*. The corresponding E declaration is illustrated below.

```
persistent collection<rel_persist> relCollect[MAXRELSTORE];
```

The names of the binary relations are stored in a persistent array of names, and is used as a dictionary to keep track of the relations. Routines for loading/storing binary relations from/on the EXODUS storage manager are provided using the *collection_scan* mechanisms provided in E.

4.3 Source Code Files and their Functionality

- `tarski.h` This file contains declarations for all the data structures used for the implementation of the Tarski interpreter and machine.
- `tarski.y` This file contains the grammar for the extended Tarski algebra specified using yacc which parses Tarski expressions and specifies actions to generate intermediate code

[8] A *collection* in E is a set of related objects that can be allocated dynamically.

for each rule in the grammar. This file implements the lexical analyzer `yylex()` and the core of the Tarski interpreter.

- `symbol.c` This file contains the routines required for creation, lookup, and installation of the symbol table.

- `init.c` This file contains definitions for the function pointers for the built-in operators of the Tarski algebra, the function `init()` installs them in the symbol table.

- `func.c` This file implements all the built-in Tarski operators, in addition to handling the creation of binary relations.

- `persistdb.h` This file contains the declarations required for persistent EXODUS storage of the binary relations. Each relation is stored as a collection of attribute pairs, and the whole database is stored as an *array* of such *collections*, and is maintained with a *simple dictionary*.

- `func_exodus.c` This file handles the persistent storage facilities using the E programming language, providing routines for storing and loading relations into/from the EXODUS storage manager.

- `code.c` This file contains the routines that execute the (intermediate) machine instructions and manipulate the execution and interpreter stacks, the function `execute()` actually *runs* the Tarski machine.

- `Makefile` This file compiles all the above files into the executable `tarskicc`, the Tarski "compiler". The files are compiled using the E compiler (an extension of g++) of EXODUS. The "compiler" can be run with command line options `-p` or `-r` for tuple sensitive tagging or relation sensitive tagging, tuple sensitive being the default.

5 Higher Level Components of IUGQL

5.1 The Graph-Tarski Translator and Optimizer

This is the main high level component of the implementation. This component translates graph oriented schemas and instances into corresponding binary relations, and translates query graphs into equivalent optimized Tarski algebra expressions using the algorithms outlined in [12, 13]. The Tarski expressions generated by the translator are then fed to the Tarski interpreter, to evaluate the final result of the query.

In this version of the translator, the input graphs (schema, instance, and query) are specified textually as a list of edges between nodes and are stored in separate files. The translator reads the input files and builds the corresponding schema/instance/query graphs. From the schema and instance graphs, commands to generate the appropriate binary relations are fed to the Tarski interpreter, which in turns generates intermediate code causing the Tarski machine to create and store the binary relations on top of the EXODUS storage manager. Of course, the loading of schema and instance graphs needs to be done only once. To work with previously created schemas and instances, it is possible to load such schemas and instances from the persistent store provided by EXODUS. The query graph is then reduced to a single node using the algorithms (*edge reduction*, *node combination*, *projection*, *chaining*, *disconnected node combination*, and *selection*) outlined in detail in [12, 13]. The Tarski expression for this single node is generated and then fed to the Tarski interpreter and machine to evaluate the final result of the query. After the query graph is evaluated, the corresponding action specified, is also performed using the algorithms outlined in [12]. The implementation of the Tarski-Graph translator and optimizer was done in C++. This greatly helped the design process as separate classes were used to model the different sub parts of the translator. In the following subsection, we list the source files and their functionality.

5.2 Source Code Files and their Functionality

- `graph.h` The header file for the *graph* class.
- `graph.c` This files implements graph manipulation functions like *add_node*, *add_edge*, *del_node*, *del_edge* etc. Graphs are internally implemented as a list of nodes and a list of edges, with a connection list for each node that stores the edges (and their connected nodes) connected to that node.
- `list.h` The header file for the *nodelist*, *edgelist*, and *clist* classes.
- `list.c` This file implements the nodelist, edgelist, and clist manipulation operations. The nodelist is implemented as a list of schema nodes with each schema node linked to a list of all the instance nodes of that type. A similar implementation is used for the edgelist. A connection list (clist) is maintained for each node that stores a list of the edges (and their connected nodes) connected to that node. Functions for manipulating these lists like *append*, *remove* etc., are implemented in this file.
- `generic_node.h` The header file for the *node*, *edge*, *c_node*, and *info* classes.
- `generic_node.c` This implements the node, edge, c_node, and info manipulation operations. Nodes and edges have information (info), labels, and other details contained in them. The info in a node/edge in turn stores the tarski_exp associated with the node/edge. The c_nodes constitute the clist. Routines for manipulation of the above information like *get_info* etc. are implemented in this file.
- `label.h` The header file for the *label* class.
- `label.c` This file implements the routines for creation and manipulation of labels for nodes/edges. The labels have a prefix and a suffix. The prefix indicates the schema, i.e.,

to which node/edge type the particular node/edge belongs, while the suffix indicates the instance number of the node/edge.

- `tarski.h` The header file for the *tarski_exp* class.
- `tarski.c` This file implements the routines for creation of basic Tarski expressions, and also for the generation of complex binary and unary Tarski expressions from basic Tarski expressions.
- `sel.h` The header file for handling the manipulation of *selected nodes* of a query.
- `sel.c` This files implements routines for maintaining a list of selected nodes, and handling projections onto selected nodes during query evaluation.
- `schem.h` The header file for handling the manipulation of *schema nodes* of a database.
- `schem.c` This file implements the handling of schema nodes needed to implement the *actions* like *node/edge addition/deletion* etc.
- `f.h` The header file for forward declaration of classes and other information.
- `main.c` This file actually implements all the algorithms (*edge reduction*, *node combination*, *projection*, *chaining*, *disconnected node combination*, *selection*[9]) needed for the translation of query graphs into equivalent Tarski algebra expressions. The implementation of the various actions is also handled in this file.
- `Makefile` This file compiles all the above files into the executable `graph2tarski`, the Graph-Tarski translator. The files are compiled using the AT&T C++ compiler CC.

The C++ header files (with the important member functions only) for the various classes are included in the appendices.

[9]Selection conditions are incorporated in the query graph when building the query graph itself, for optimization reasons.

5.3 Algorithms Used for Implementation of the Translator

In this subsection, we briefly discuss the implementation (with pseudo-code) of the main algorithms [12, 13] used in the translation of query graphs into equivalent Tarski algebra expressions.

5.3.1 Edge Reduction

Edge reduction replaces multiple edges between a pair of nodes by a single equivalent edge between the nodes, that represents the intersection of all the edges. The implementation of edge reduction is fairly straightforward.

```
reduce_edges(graph)

begin
  while (e1 = graph.get_edge()) do        /* get any edge */
      n1 = e1.from_node();
      n2 = e1.to_node();
      t1 = get_tarski_exp(e1);
      graph.del_edge(e1);

      while (e2 = graph.get_edge(n1, n2)) do
            /* get other edges between n1 and n2 */
            t2 = get_tarski_exp(e2);
            t1 = tarski_exp(intersect(t1,t2));
            graph.del_edge(e2);
      end;

      graph.add_edge(n1, n2, t1);
            /* add new edge between n1 and n2 with tarski_exp t1 */
  end;
end;
```

5.3.2 Node Combination

Node combination combines two nodes connected by a single edge into one single node, and simulates the edges leaving/entering the two nodes by appropriate compositions. The tricky issue involved in the node combination is handling the projections onto the selected nodes after the node combination. The handling of projections is crucial towards the evaluation of the final result of the query. We describe the algorithm used for projection in the next subsection.

```
combine_nodes(graph, e, n1, n2) /* combine nodes n1 and n2 */
                                /* connected by edge e from n1 to n2 */
begin
  t1 = get_tarski_exp(e);
  t2 = tarski_exp(tau, t1);
  newnode = graph.add_node(t2);
   /* add new node with tarski_exp t2 = tau(t1) */

  while (e1 = graph.get_edge(n1)) do
      /* handle edges entering/leaving node n1 */
      t3 = tarski_exp(ltag, t1);  /* or inv(ltag(t1)) */
      graph.add_edge(n1, e1.other_node(), t3);
  end;

  while (e2 = graph.get_edge(n2)) do
      /* handle edges entering/leaving node n2 */
      t4 = tarski_exp(rtag, t1);  /* or inv(rtag(t1)) */
      graph.add_edge(n2, e2.other_node(), t4);
  end;

  update_projections(graph, n1, n2, newnode, t1);
  /* update projections onto selected nodes */

  graph.del_edge(e);
  graph.del_nodes(n1, n2);
end;
```

5.3.3 Projection

As discussed in [12, 13], it is necessary to *backtrack* the sequence of node combinations in order to compute the final result of the query. This is accomplished in the implementation by an *incremental* update of the projections at each step of the node combination. Initially, a list of the selected nodes is maintained. Every time, a node combination (involving selected nodes or bold[10] nodes) is performed, the current projection onto the selected nodes and the current bold node that projects onto the selected node are stored. This provides a mechanism to update the projections onto the appropriate selected nodes if the bold nodes are subsequently involved in other node combinations. Finally, when the algorithm terminates, the selected nodes will have the appropriate projections. In general, the result may contain duplicates, which may be eliminated as argued in [12]. Duplicate elimination can be simulated in the Tarski algebra, but can be supported much more efficiently as a primitive in an implementation [12].

```
update_projections(graph, n1, n2, newnode, t1); /* newnode is a result of */
                                                /* node combining n1 and n2 */
                                                /* t1 is the tarski_exp of the */
begin                                           /* edge connecting n1 and n2 */
  if (n1.is_selected())
     begin
       n1.boldnodename = newnode.name;
       n1.projection = tarski_exp(ltag, t1);
     end;

  if (n2.is_selected())
     begin
       n2.boldnodename = newnode.name;
       n2.projection = tarski_exp(rtag, t1);
     end;

  if (n1.is_bold())
```

[10] A bold node is one that is a result of a previous node combination [12, 13].

```
    begin
      for each selectednode (with boldnodename = n1.boldnodename) do
          selectednode.boldnodename = newnode.name;
          selectednode.projection =
                  tarski_exp(compose, ltag(t1), selectednode.projection);
                  /* Update projection path */
      end;
    end;

  if (n2.is_bold())
    begin
      for each selectednode (with boldnodename = n2.boldnodename) do
          selectednode.boldnodename = newnode.name;
          selectednode.projection =
                  tarski_exp(compose, rtag(t1), selectednode.projection);
                  /* Update projection path */
      end;
    end;
end;
```

5.3.4 Chaining

This is a very important query optimization technique where a chain of edges between two nodes is replaced by a single edge. Again, we have adopted an incremental algorithm for chaining to make the implementation clean and easy to comprehend. We identify "*possible to chain nodes*" as those that are *not selected* and have only *two* edges entering/leaving them.

```
remove_chains(graph)
begin
  if (chain_node = graph.possible_to_chain())
    begin
      e1 = graph.get_edge(chain_node);
      e2 = graph.get_edge(chain_node);
      t1 = get_tarski_exp(e1);
      t2 = get_tarski_exp(e2);
      n1 = one node at end of chain;
      n2 = other node at other end of chain;
      t3 = tarski_exp(compose, t1, t2); /* or inv(t1) or inv(t2) */
                                        /* depending on edges leaving/entering */
```

```
       graph.add_edge(n1, n2, t3);
       graph.del_edges(e1, e2);
       graph.del_node(chain_node);
     end;
end;

possible_to_chain()
begin
  while (n1 = graph.get_node()) do
    if (n1.is_not_selected() and (num_edges(n1) == 2))
       return n1;
    else
       repeat while;
  end;
  return NULL;
end;
```

5.3.5 Disconnected Node Combination

This is very similar to a regular node combination of two nodes connected by an edge except that we create an imaginary edge between the two disconnected nodes to reflect the cartesian product of all possible combinations of the two nodes[11]. After the creation of this imaginary edge, node combination and projection are done in a similar fashion as detailed previously.

5.3.6 Overall Translation Algorithm

The overall translation algorithm is implemented using the algorithms illustrated in the above subsections. First the schema, instance, and query graphs are built from the input files. Then the query graph is translated into an equivalent Tarski algebra expression, which is fed to the Tarski interpreter and machine to provide the final answer to the query.

```
graph2tarski_translate()
begin
```

[11]The semantics of disconnected nodes is exactly captured by the cartesian product, and this can be done in the Tarski algebra, for details see [12, 13].

```
  build_schema_graph();
  build_instance_graph();
  build_query_graph();

  reduce_edges(query_graph);
  remove_chains(query_graph);

  while(e = query_graph.get_edge()) do
    begin
      combine_nodes(query_graph, e, e.from_node(), e.to_node());
      /* combine_nodes invokes update_projections(...); */
      reduce_edges(query_graph);
      remove_chains(query_graph);
    end;

  combine_disconnected_nodes(query_graph);
  perform_actions(schema_graph, query_graph);
  communicate_IUGQL_lower_level();
end;
```

6 Using IUGQL

In this version of the implementation, graphs are input as a list of edges between nodes in text files. Three files are used for inputting schema, instance, and query graphs, and are appropriately named *schema*, *instance*, and *query*[12] respectively. The Graph-Tarski translator takes these three files as input from the command line with an option of `-b` to build the schema and instance graphs if needed. If the `-b` option is not specified, previously created schema and instance graphs are loaded from EXODUS. Due to some problems with the g++ compiler (which was needed to compile the lower level Tarski machine to provide persistence), the translator had to be compiled with AT&T C++. Therefore, the two levels could not be compiled into one executable module. To work around this problem, the

[12]The query file specifies both the query graph and the action.

communication between the translator and interpreter was accomplished through files (that are transparent to the user). The translator generates Tarski expressions and writes them into a file which is used by the interpreter as input. To make all this transparent to the user, the commands

```
graph2tarski (-b) schema instance query
tarskicc (-p) (-r)
```

are put into a single file named *iugql* with execute permissions and the system can be run with just one command *iugql*. Currently, the system is a working prototype and is being tested extensively. This, we feel, is a good example of an implementation of an object based graph oriented query language with a strong algebraic foundation.

7 Examples

In this section, we show actual examples to input schema, instance, and query graphs to IUGQL, and illustrate the output generated by IUGQL. This section can be used a tutorial introduction to IUGQL. It must be noted that in all the query outputs we illustrate, the first relation is the relation corresponding to the final node that the query graph reduces to, and the subsequent relations are the new/updated relations that the actions specify.

7.1 Schema Input File

We present below a schema of a persons database. This is a textual representation of the schema graph in Figure 1 (top), where each line represents an edge (with from/to nodes).

```
graphtype:schema
SP_1 c_1 P_1          /* Edge c_1 connects node SP_1 to node P_1 */
P_2 ch_1 SP_2
```

```
P_2 n_1 S_1""          /* Simple node S_1 with no value */
P_2 g_1 S_2""
P_2 age_1 S_3""
end
```

7.2 Instance Input File

We present below an instance based on the persons schema. This is a textual representation of the instance graph in Figure 1 (bottom). A tabular form of the same instance is illustrated in Figure 4.

```
graphtype:instance
SP_1 c_1 P_1
SP_1 c_2 P_2
SP_1 c_3 P_3
SP_1 c_4 P_4
SP_1 c_5 P_5
SP_1 c_6 P_6
SP_1 c_7 P_7
SP_2 c_8 P_3
SP_2 c_9 P_4
SP_4 c_10 P_5
SP_4 c_11 P_6
SP_5 c_12 P_7
P_1 n_1 S_1"Brian"     /* Simple node S_1 with value Brian */
P_1 ch_1 SP_2
P_2 ch_2 SP_2
P_2 n_2 S_2"Glenda"    /* Simple node S_2 with value Glenda */
P_3 ch_3 SP_3
P_4 n_4 S_41"Jim"      /* Simple node S_41 with value Jim */
P_4 g_4 S_42"Male"     /* Simple node S_42 with value Male */
P_4 ch_4 SP_4
P_5 n_5 S_51"Cindy"    /* Simple node S_51 with value Cindy */
P_5 g_5 S_52"Female"   /* Simple node S_52 with value Female */
P_5 ch_5 SP_5
P_6 n_6 S_61"Jim"      /* Simple node S_61 with value Jim */
P_6 g_6 S_62"Male"     /* Simple node S_62 with value Male */
P_6 ch_6 SP_6
```

```
P_7 n_7 S_71"Cindy"    /* Simple node S_71 with value Cindy */
P_7 g_7 S_72"Female"   /* Simple node S_72 with value Female */
end
```

7.3 Query Input File - Node Addition

We present below a query graph that specifies parent-child pairs with the same name, and a node addition of a new node PC specifying such parent-child pairs, and the corresponding parent and child edges. For the given instance, this query should return the pairs {P4,P6} and {P5,P7}.

```
graphtype:query
P$1 ch_1 SP_1       /* $ between prefix and suffix indicates selected node */
SP_1 c_1 P$2
P_1 n_1 S_1""       /* Simple node S_1 with no value */
P_2 n_2 S_1""
end
NA                  /* Action node addition */
PC_1 par_1 P_1
PC_1 child_1 P_2
end
```

7.3.1 Sample Run - Node Addition

We present below a sample run of the system on the persons instance, the above query, and the node addition action. We see that the output corresponds to what is expected, i.e., the pairs {P4,P6} and {P5,P7}.

```
Chaining edges ch1 and c1
Chaining edges n1 and n2
Node combining nodes P1 and P2
Final query graph
-----------------------------------------------------
NODE : label is tau(inter(compose(ch,c),compose(n,inv(n))))_P1P2
tarski_exp : tau(inter(compose(ch,c),compose(n,inv(n))))
```

```
ADD NEW RELATION PC (FOR NA)
ADD NEW RELATION par (FOR EDGE IN NA)
ADD NEW RELATION child (FOR EDGE IN NA)

     [[[ch C c] & [n C [n I]]] Tau]
-------------------------------------------------------------------
[[ch C c] & [n C [n I]]]-Tau                  P-Tau
-------------------------------------------------------------------
[P4][P6]                                      [P4][P6]
[P5][P7]                                      [P5][P7]

                                  PC
-------------------------------------------------------------------
[[ch C c] & [n C [n I]]]-Tau                  P-Tau
-------------------------------------------------------------------
[P4][P6]                                      [P4][P6]
[P5][P7]                                      [P5][P7]

                                 par
-------------------------------------------------------------------
[[ch C c] & [n C [n I]]]-L                    P
-------------------------------------------------------------------
[P4][P6]                                      P4
[P5][P7]                                      P5

                               child
-------------------------------------------------------------------
[[ch C c] & [n C [n I]]]-R                    P
-------------------------------------------------------------------
[P4][P6]                                      P6
[P5][P7]                                      P7
```

7.4 Query Input File - Edge Addition

We present below a query graph that specifies pairs of persons, one being the grandparent of the other, and an edge addition of a new grandparent edge between such person nodes. For the given instance, this query should return the pairs {P1,P5}, {P2,P5}, {P1,P6}, {P2,P6},

and {P4,P7}.

```
graphtype:query
P$1 ch_1 SP_1       /* $ between prefix and suffix indicates selected node */
SP_1 c_1 P_2
P_2 ch_2 SP_2
SP_2 c_2 P$3
end
EA                  /* Action edge addition */
P_1 gpar_1 P_3
end
```

7.4.1 Sample Run - Edge Addition

We present below a sample run of the system on the persons instance, the above query, and the grandparent edge addition action. We see that the output corresponds to what is expected, i.e., the pairs {P1,P5}, {P2,P5}, {P1,P6}, {P2,P6}, and {P4,P7}.

```
Chaining edges c1 and ch2
Chaining edges ch1 and compose(c,ch)c1ch2
Chaining edges c2 and compose(ch,compose(c,ch))ch1compose(c,ch)c1ch2
Node combining nodes P3 and P1
Final query graph
-----------------------------------------------------
NODE : label is tau(compose(inv(c),inv(compose(ch,compose(c,ch)))))_P3P1
tarski_exp : tau(compose(inv(c),inv(compose(ch,compose(c,ch)))))

ADD NEW RELATION gpar (FOR EA)

[[[c I] C [[ch C [c C ch]] I]] Tau]
-------------------------------------------------------------------
[[c I] C [[ch C [c C ch]] I]]-Tau            P-Tau
-------------------------------------------------------------------
[P5][P1]                                     [P5][P1]
[P5][P2]                                     [P5][P2]
[P6][P1]                                     [P6][P1]
[P6][P2]                                     [P6][P2]
[P7][P4]                                     [P7][P4]
```

gpar

P	P
P1	P5
P2	P5
P1	P6
P2	P6
P4	P7

7.5 Query Input File - Node Deletion

We present below a query graph that specifies persons who have children, and a node deletion that deletes all such parents from the database. For the given instance, this query should return the persons P1, P2, P4, and P5.

```
graphtype:query
P$1 ch_1 SP_1       /* $ between prefix and suffix indicates selected node */
SP_1 c_1 P_2
end
ND                  /* Action node deletion */
P_1
end
```

7.5.1 Sample Run - Node Deletion

We present below a sample run of the system on the persons instance, the above query and the parent node deletion action. Node deletion also updates the edges connected to the deleted node. In the output, we first see the relation that corresponds to the final node after the translation, and then see the results of updating the node and edge relations to reflect the removal of these persons from the database.

```
Chaining edges ch1 and c1
Node combining nodes P1 and P2
Final query graph
--------------------------------------------------
```

```
NODE : label is tau(compose(ch,c))_P1P2
tarski_exp : tau(compose(ch,c))

UPDATE RELATION P (FOR ND)
UPDATE TO EDGE RELATION c (FOR ND)
UPDATE FROM EDGE RELATION ch (FOR ND)
UPDATE FROM EDGE RELATION n (FOR ND)
UPDATE FROM EDGE RELATION g (FOR ND)
UPDATE FROM EDGE RELATION age (FOR ND)
```

[[ch C c] Tau]

[ch C c]-Tau	P-Tau
[P1][P3]	[P1][P3]
[P1][P4]	[P1][P4]
[P2][P3]	[P2][P3]
[P2][P4]	[P2][P4]
[P4][P5]	[P4][P5]
[P4][P6]	[P4][P6]
[P5][P7]	[P5][P7]

[P D [[[ch C c] L] RId]]

P-l	P-r
P3	P3
P6	P6
P7	P7

[c D [[[[[ch C c] L] RId] C [c I]] I]]

SP	P
SP1	P3
SP1	P6
SP1	P7
SP2	P3
SP4	P6
SP5	P7

```
   [ch D [[[[ch C c] L] RId] C ch]]
-------------------------------------------------------------------
P                                                  SP
-------------------------------------------------------------------
P3                                                 SP3
P6                                                 SP6

     [n D [[[[ch C c] L] RId] C n]]
-------------------------------------------------------------------
P                                                  n
-------------------------------------------------------------------
P6                                                 Jim
P7                                                 Cindy

     [g D [[[[ch C c] L] RId] C g]]
-------------------------------------------------------------------
P                                                  g
-------------------------------------------------------------------
P6                                                 Male
P7                                                 Female

 [age D [[[[ch C c] L] RId] C age]]
-------------------------------------------------------------------
P                                                  age
-------------------------------------------------------------------
```

7.6 Query Input File - Edge Deletion

We present below a query graph that specifies persons who have children and their set of children, and an edge deletion that deletes the child edge from all such parents. For the given instance, this query should return the pairs {P1,SP2}, {P2,SP2}, {P4,SP4}, and {P5,SP5}.

```
graphtype:query
P$1 ch_1 SP$1       /* $ between prefix and suffix indicates selected node */
SP_1 c_1 P_2
end
ED                  /* Action edge deletion */
P_1 ch_1 SP_1
```

```
end
```

7.6.1 Sample Run - Edge Deletion

We present below a sample run of the system on the persons instance, the above query, and the child edge deletion action. Again here, the first relation in the output corresponds to the final node after the translation and the other relations show the results of updating the child edge relation. The output corresponds exactly to what is expected.

```
Node combining nodes P1 and SP1
Node combining nodes tau(ch)P1SP1 and P2
Final query graph
--------------------------------------------------
NODE : label is tau(compose(rtag(ch),c))_tau(ch)P1SP1P2
tarski_exp : tau(compose(rtag(ch),c))

UPDATE RELATION ch (FOR ED)

                  [[[ch R] C c] Tau]
------------------------------------------------------------------
[[ch R] C c]-Tau                              P-Tau
------------------------------------------------------------------
[[P1][SP2]][P3]                               [[P1][SP2]][P3]
[[P1][SP2]][P4]                               [[P1][SP2]][P4]
[[P2][SP2]][P3]                               [[P2][SP2]][P3]
[[P2][SP2]][P4]                               [[P2][SP2]][P4]
[[P4][SP4]][P5]                               [[P4][SP4]][P5]
[[P4][SP4]][P6]                               [[P4][SP4]][P6]
[[P5][SP5]][P7]                               [[P5][SP5]][P7]

                                  ch
------------------------------------------------------------------
P                                             SP
------------------------------------------------------------------
P3                                            SP3
P6                                            SP6
```

8 Summary and Future Work

IUGQL is a good example of an object based query language with a strong underlying algebraic foundation. IUGQL is currently a working prototype in the database lab at Indiana University.

In the next version of the implementation, we propose to incorporate the creation and use of indices to manipulate binary relations to make the Tarski machine more efficient. We also plan to add an X-Window based graphical front end to IUGQL to enable users to actually queries graphically. In this version, graphs are input textually. We also plan to incorporate the translation of query graphs into the extended Tarski algebra [14], to further optimize query evaluation [14]. We are currently working on extending the graph oriented query language itself to allow the specification of more complex generalized conjunctive declarative queries. Also, in this version, composition of actions is not supported, i.e., the effects of one action cannot be used to influence subsequent actions. The system can be fairly easily extended to support composition of actions, and we are currently working on adding this feature to the system.

9 Acknowledgements

The authors would like to thank K.U. Srinivasan and Uday Naik for help with the implementation of the graph primitives, and for many crucial design discussions during the implementation of Graph-Tarski translator.

References

[1] F. Bancilhon. Object Oriented Database Systems. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, Austin, Texas, 1988, pp. 152–

162.

[2] C. Beeri. New Data Models and Languages - The Challenge. In *ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, San Diego, California, 1992, pp. 1–15.

[3] M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.T. Schuch, E.J. Shekita, and S.L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In *Readings in Object-Oriented Database Systems.* Edited by S.B. Zdonik and D. Maier, Morgan Kaufmann Publ., 1989.

[4] Exodus Project Group, Univ. of Wisconsin. Introduction to GNU E, 1992.

[5] G. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Austin, Texas, 1985, pp. 268–279.

[6] M. Gyssens, J. Paredaens, and Dirk Van Gucht. A Graph Oriented Object Database Model. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, Nashville, Tenn., 1990, pp. 417–424.

[7] M. Gyssens, L. Saxton, and Dirk Van Gucht. Tagging as an Alternative to Object Creation. Presented at *The Dagstuhl Seminar on Query Processing in Object Oriented, Complex Object, and Nested Relation Databases.*

[8] B.W. Kernighan and R. Pike. The UNIX Programming Environment. *Prentice Hall Software Series*, 1984.

[9] B.W. Kernighan and D.M. Ritchie. The C Programming Language, 2nd Edition. *Prentice Hall Software Series*, 1988.

[10] S. Khoshafian and P. Valduriez. Sharing, Persistence, and Object Orientation: A Database Perspective. Advances in Database Programming Languages. *ACM Press, Frontier Series*, pp. 221–240.

[11] S.B. Lippman. C++ Primer, 2nd Edition. *Addison-Wesley Publishing Company*, 1991.

[12] V. Sarathy, L. Saxton and D. Van Gucht. Translating Query Graphs into Tarski Algebra Expressions. *Technical Report # 342*, Dept. of Computer Science, Indiana University, December 1991.

[13] V. Sarathy, L. Saxton and D. Van Gucht. Algebraic Foundation and Optimization for Object Based Query Languages. To appear in *Proc. Ninth IEEE Int'l Conf. on Data Engineering*, Vienna, Austria, 1993.

[14] V. Sarathy, L. Saxton and D. Van Gucht. An Object Based Algebra for Parallel Query Processing and Optimization. *Technical Report # 368*, Dept. of Computer Science, Indiana University, December 1992.

[15] B. Stroustrup. The C++ Programming Language, 2nd Edition. *Addison-Wesley Publishing Company*, 1991.

[16] A. Tarski. On the Calculus of Relations. *Journal of Symbolic Logic*, 6, 1941, pp. 73–89.

Appendices
Class Declarations for the Graph-Tarski Translator

A graph.h

```
class graph
{
private:
 graphtype gtype;
 nodelist nl;         /* a list of schema nodes */
 edgelist el;          /* a list of schema edges */

public:
 graph();              /* constructor */
 graph(graphtype g);   /* constructor */
 ~graph();             /* destructor */
 void print();
 node*  add_node(label l, info* ptr, boolean& Exists);
 /* add a node to the graph if it does not exist already */

 edge*  add_edge(label l, info* ptr, node* n1, node* n2, boolean& Exists);
 /* add an edge between n1 & n2 to the graph if it does not exist already */

 void del_node(node* ptr); /* delete a node from the graph */
 void del_edge(edge* ptr); /* delete an edge from the graph */

 node* get_node(char* name, int visit, boolean set);
 /* get a node whose label is name, visit_num is less than visit.
    if name is NULL, return a random node.
    if SET is TRUE it sets the visit_num of the node to visit. */

 edge* get_edge(node* n1, node* n2, int visit, boolean set);
 /* get an edge between n1 and n2. if n1 is NULL get any edge containing
 n2, similarly if n2 is NULL. if both are NULL return any random edge.
 get a random edge whose visit_num is less than visit. if SET is TRUE the
 visit_num of the edge is set to visit. */

 graphtype get_graphtype();
```

```
void mark_node(node* ptr);
void mark_edge(edge* ptr);
void unmark_node(node* ptr);
void unmark_edge(edge* ptr);
void unmark_alledge();
void unmark_allnode();

int get_num_edges(node* nptr); /* return number of edges to and from node */

boolean check_if_edges(node* nptr); /* check if the node has any edges */

node* possible_to_chain(boolean& single); /* check for "chainability" */
```

B list.h

```
class nodelist
{
private:
 node* head;
 node* last;

public:
 nodelist();                        /* constructor */
~nodelist();                        /* destructor */
 node* append(info* i , label l);   /* append a new node to the nodelist */
 void remove(node* n);              /* remove a node from the nodelist */
 node* head_node();                 /* return head_node of the nodelist */
 node* next(node* ptr);             /* return next node in the nodelist */
 node* search(label l);             /* search for a specific node */
};

class edgelist
{
private:
 edge* head;
 edge* last;

public :
 edgelist();                        /* constructor */
 ~edgelist();                       /* destructor */
 edge* append(node* from, node* to, info* i , label l);/* append to edgelist */
 void remove(edge* n);              /* remove an edge from the edgelist */
 edge* head_edge();                 /* return head_edge of the edgelist */
 edge* next(edge* next);            /* return next node in the edgelist */
 edge* search(label l);             /* search for a specific edge */
};

class clist
{
private:
 c_node* head;
 c_node* last;

public :
```

```
 clist();                           /* constructor */
 ~clist();                          /* destructor */
 c_node* append(edge* e, node* n);  /* append a node and edge to the c_list */
 void remove(c_node* n);            /* remove a node from the c_list */
 c_node* head_node();               /* return head_node of the c_list */
 c_node* next(c_node* ptr);         /* return next c_node */
 clist* search(node* n, edge* e);   /*  return a list of c_nodes satisfying
                                       search criteria. if n is NULL search
                                       only for e and vice versa */

};
```

C generic_node.h

```
class info        /* info stored in nodes and edges */
{
private:
 char* value;

public:
 tarski_exp* tarski;                 /* tarski_exp of the edge/node */
 objtype typ;                        /* complex, simple, or bold */
 boolean selected;                   /* selected node or not */
 info (tarski_exp* tarski,objtype x); /* constructor */
 info (tarski_exp* tarski, objtype x, boolean sel);
 info (tarski_exp* tarski, objtype x, char* baseval, boolean sel);
      /* constructors for selected nodes */
 info ();                            /* constructor */
 ~info ();                           /* destructor */
 void set_selected(boolean flag);    /* set info to selected or not */
 char* get_str_data();
 void set_str_data (char* s);        /* data manipulation functions */
};

class node
{
friend class nodelist;
friend class graph;

private:
 node* next;
 boolean del;           /* delete flag */
 clist connection;      /* a list of nodes connected to the node by an edge */
 boolean is_schema;     /* schema node or not */
 nodelist ilist;        /* a ptr to the instance list */
 node* sch_node;        /* a ptr to the schema node */
 label id;              /* a label which is unique to the node */
 info* data;            /* info stored in the node */
 boolean mark;          /* used to mark a node */

public:
 node(info* i, label id);/* constructor */
 node();                /* constructor */
```

```
 ~node();                /* destructor */
 label get_label();      /* get node's label */
 info*  get_info();      /* return a ptr to a copy of the node's info */
 void set_label(label l);/* set node's label to l */
 void set_info(info* i); /* makes a copy of *i and puts it in node */
 void set_selected(boolean flag); /* sets a node to be selected or not */
 boolean is_deleted();   /* checks if node is deleted or not */
 boolean is_selected();  /* checks if node is selected or not */

class edge
{
friend class edgelist;
friend class graph;

private:
 edge* next;
 boolean del;            /* delete flag */
 boolean is_schema;      /* schema edge or not
 edgelist ilist;         /* a ptr to the instance list */
 edge* sch_edge;         /* a ptr to the schema edge */
 label id;               /* a label which is unique to the edge */
 info* data;             /* info stored in the node */
 node* from;             /* from node of edge */
 node* to;               /* to node of edge */
 boolean mark;           /* used to mark the node */

public:
 edge (node* f, node* t, info* i, label id); /* constructor */
 edge();                 /* constructor */
 ~edge();                /* destructor  */
 node* from_node();      /* get the from node of edge */
 node* to_node();        /* get the to node of edge */
 label get_label();      /* get label of edge */
 info*  get_info();      /* return a ptr to the copy of the edge's info */
 void set_label(label l);/* set label of edge to l */
 void set_info(info* i); /* makes a copy of *i and puts it in edge
 boolean is_deleted();   /* checks if edge is deleted or not */
};

class c_node
{
```

```
friend class clist;

private:
 c_node* next;
 boolean del;

public:
 node* n;                /* node in this c_node */
 edge* e;                /* edge in this c_node */
 c_node(edge* e1, node* n1); /* constructor */
 c_node() ;              /* destructor */
 boolean is_deleted();   /* checks if c_node is deleted or not */
};
```

D label.h

```
class label
{
private:
 char* prefix;           /* schema part of label */
 char* suffix;           /* instance part of label */

public:
 label(char* p, char* s);/* constructor */
 label();                /* constructor */
 ~label();               /* destructor */
 void set_p(char* p);    /* set prefix to p */
 void set_s(char* s);    /* set suffix to s */
 char* get_p();          /* get prefix */
 char* get_s();          /* get suffix */
 char* get_l();          /* get whole label */
};
```

E tarski.h

```
class tarski_exp
{
private:
 char* value;
public:
 tarski_exp(char* l);    /* constructor */
 tarski_exp();           /* constructor */
 ~tarski_exp();          /* destructor */
 tarski_exp& generate_if(char* op, tarski_exp& t1, tarski_exp& t2,
                         tarski_exp& t3);
 /* generate an if/nif tarski expression depending on operation op */

 tarski_exp& generate2(char* op, tarski_exp& t1, tarski_exp& t2);
 /* generate a new tarski expression depending on operation op for
    binary operators */

 tarski_exp& generate1(char* op, tarski_exp& t);
 /* generate a new tarski expression depending on operation op for
    unary operators */

 tarski_exp& generate_asgn(char* assgn, tarski_exp& t);
 /* generate assignment expression */

 tarski_exp& generatesel(char* op, char* cnst, tarski_exp& t);
 /* generate a new tarski expression for left and right selection */

 char* get();             /* get tarski_exp as a character string */
};
```