Graph Transformations based on Relational Databases

Master Thesis
Aachen, August2008

Submited to
Department of Computer Science 3
Prof. Dr.-Ing. M. Nagl
RWTH Aachen

Written By

Qasim Ali

Born at Rawalpindi, Pakistan.

Reviewer:	University professor Dr. -Ing. M. Nagl
	University professor Dr. rer.nat. O. Spaniol
Supervisor:	Dipl.-Inform. E. Weinell

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

Lehrstuhl für
Informatik 3

Department of
Computer Science 3

Author's Declaration

Herby, I declare that I have written this thesis autonomously and I have not used other resources than indicated.

This thesis is not presented at another institution and has not been published.

———————————————

Qasim Ali

Aachen , August 04, 2008

Acknowledgments

This thesis is the final research work required for completion of my master's degree in software systems engineering at RWTH Aachen. During my studies in this institution, I have got an esteemed knowledge and have been familiarized with professional skills, which I believe will always provide me a healthy support in my future endeavors.

First and most of all I would like to thanks my parents for their great support through out my academic career. Their encouragement, love and support have always driven me toward better approach of execution and produce my level best.

This thesis would not have been possible without the support of my supervisor Mr. Erhard Weinell. I would like to thank him for his guidance, advice and encouragement throughout the course of this thesis. I have benefited enormously from his valuables insights and mentoring. Despite his tight schedule, he gave countless suggestions to improve the quality of my thesis.

I would also like to convey thanks to the Department of Computer Science 3 (Software Engineering) and its head Professor Dr.-Ing. M. Nagl for providing academic support and laboratory facilities.

Abstract

After years of research graph transformations have evolved and acquired a matured state. Graph transformations are used in various application areas such as rule based image recognition, translation of diagram languages, model driven software development, service-oriented Applications, pattern recognition, semantics of programming languages, implementation of functional programming specification of database systems, specification of abstract data types, etc. This thesis; however, explores the idea of supporting graph transformations based on relational databases.

The approach discussed includes a query and transformation language (QTL). This transformation language can act as a platform to build graph applications using graph transformations. QTL can easily be extended with new constructs and can serve as a base layer for domain-specific applications. By acting as a base layer for other graph languages, it allows developers to work at a higher level functionality provided rather than earlier approach of developing a new code generation modules or interpreter for a new application.

The work is focused on adding graph transformation support in DRAGOS graph database. DRAGOS eases the development of graph based applications by providing a uniform graph-oriented data storage facility using a relational database system as backend storage. DRAGOS so far, supports various backend relational databases such as MySQL, PostgreSQL, Derby etc. The standard operations provided by the relational database's underlying data manipulation language can be used to implement graph transformation engine. This thesis exploits this very support to add a Query and Transformation Mechanism (QTM) in DRAGOS. QTM adds support for graph transformations in DRAGOS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph transformation originally evolved in reaction to shortcomings in the expressiveness of classical approaches to rewriting, like Chomsky grammars[35] and term rewriting, to deal with non-linear structures. The first proposals, appearing in the late sixties and early seventies according to [6] were concerned with rule based image recognition, translation of diagram languages, etc. In all of these years, graph transformations have been successful in approaching solutions of various challenges related to software engineering. They have been used in diverse application areas from model driven software development, service-oriented applications to pattern recognition techniques[5]. There are already a wide range of tools available which use graph transformation in area of simulation or verification like AGG[37], AToM3[12], DiaGen[25], FUJABA[29], GReAT, GROOVE[31] and VIATRA[10].

Another application of graph transformations is to model queries on a high abstraction level[32]. During the past years, intensive research has been done on use of graph transformation as a visual query and data manipulation language for relational databases. A brief selection of work in this field is given as follows:

- Andries and Engels proposed in [3, 4], a hybrid database query language (HQL/EER)based upon graph transformation. HQL has been discussed in detail in section 6.1 of this document.

- In [40], Gergely Varró, recently, presented an approach to implement a graph transformation engine as an EJB3-specific plugin by using EJB QL queries for pattern matching. The essence of his approach is to create an EJB QL query for the precondition of each graph transformation rule. Pattern matching and updating phases of a rule application are implemented in a public method of a stateless session bean by executing the prepared EJB QL query and by manipulating persistent objects, respectively.

- Graph database GRACE [36] includes a graph query language called SAFARI, which supports both attribute and structural searches.

- GraphLog [9] is a visual graph query language. GraphLog queries ask for patterns that must be present or absent in the database graph. Each such pattern, called a query graph, defines a set of new edges (i.e., a new relation) that are added to the graph whenever the pattern is found.

> GraphLog queries are sets of query graphs, called graphical queries. An edge used in a query graph either represents a base relation or is itself defined in another query graph. In 1995, Frank Ch. Eigler proposed a mechanism of translating GraphLog in to SQL[15].

It is common in all these approaches that they investigate how graph transformations can contribute to database management systems and tasks. However, it was PROGRES (PROGRAMMED GRAPH REWRITING SYSTEMS)[27] which represented the first major application of graph transformation in software engineering[17]. PROGRES is a language and a tool for programmed graph rewriting Systems. It performs graph transformation by rewriting graph systems. It offers language constructs for defining graph schema and graph transformation rules. It uses directed, attributed, node and edge labeled graphs as underlying data model. PROGRES uses GRAS as graph storage and it is used at the runtime of the graph transformation system.

PROGRES offers a developing environment to work which includes an editor with an integrated analyzer which points out all violations against the stated semantics of the language. An interpreter is also included which supports the incremental execution of a specification and presents the resulting graph structure in a graphical view[7]. Finally, a compiler translates the specification into adequate and efficient code. The graph schema and the actual graph are saved in GRAS(GRAPH Storage [20]). PROGRES interpreter operate on these stored graphs. In other words, GRAS is used at run time by the graph transformation systems. GRAS(GRAPH Storage [20]) has been succeeded by DRAGOS.

DRAGOS (Database Repository for Applications using Graph Oriented Storage) is a graph oriented database management system. It uses relational database system as back-end for storing graphs. DRAGOS supports various back-end relational databases such as MySQL, PostgreSQL, Derby etc. GRAS and DRAGOS system have been developed at the department of Computer Science 3 RWTH Aachen University. Unlike, GRAS, DRAGOS has been designed on a multi-layered architecture. This makes extensions in DRAGOS much easier. GRAS was not platform independent while DRAGOS's implementation has been done in java which makes it platform independent. DRAGOS offers exchangeable storage back-ends. This gives user lot of options to choose a storage mechanism of its choice. It provides a fast in-memory version for testing. DRAGOS is discussed in detail in section 2.2. Currently, implementation of graph transformation system in DRAGOS is only possible by using PROGRES. The code generated by PROGRES can be used by applications using DRAGOS to support graph transformations. This deployment procedure, however, is quite complicated.

This thesis discusses an approach of supporting graph transformations in DRAGOS involving a query and transformation language (QTL). The graph transformation language can act as a platform to build graph applications using graph transformations. QTL can easily be extended with new constructs and can serve as a base layer for domain-specific applications. QTL has been discussed in detail in section 3.4. Along with QTL, DRAGOS's support for relational databases can be exploited to develop a transformation system.

As already discussed DRAGOS supports various back-end relational databases. Standard Query Language (SQL) is the standard data access and manipulation language for relational databases. The standard operations provided by SQL can be used to implement graph transformation engine. This thesis exploits this support to add a graph transformation system in DRAGOS. The developed system out-performs the traditional system developed by using PROGRES by a huge margin. The results of runtime experiments conducted to compare both system are part of this document. There were two ways of developing graph transformation system in DRAGOS, they have been discussed in next sections.

1.1 Graph Transformation System in DRAOGS

Supporting graph transformations in DRAGOS requires development of a query and transformation mechanism which should represent a graph transformation system using a basic query and transformation language (QTL). The details of QTL can be seen in detail in chapter 3.4. Query and transformation language of DRAGOS is named DRAGULA [43], which is an abbreviation of DRAGOS Unified Language. There are two possibilities of executing graph transformations in DRAGOS.

1. Universal Solution
2. Backend Specific Solution

The following sections describe pros and corns of both approaches.

1.1.1 Universal Solution

The DRAGOS API provides atomic retrieval and update operations. First possible solution is to create a solution using these operations. This approach requires, a transformation manager in DRAGOS, as shown in figure 1.1, which accepts a pattern created by the application in QTL. Transformation Manager parses this pattern. The result is constructed by using atomic operations provided by DRAGOS‘s core services. One important characteristic of this approach, as shown in figure 1.1, is that the operations provided by core services use DRAGOS database implantations to access the underlying storage. Usually, one operation makes a single database access which is usually one SQL statement at background. So result is prepared after execution of several SQL statements or database accesses. Prepared result is then given back to application. This solution is called Universal due to its back-end independence. This back end independence suggests that this solution is valid for in-memory version and for all other RDBMS‘s supported by DRAGOS.

Universal solution, as discussed above relies on the DRAGOS core API to made database accesses. Each operation of core API executes at least one SQL statement. So in order to compile results for a complex pattern, a lot of atomic operations are required, subsequent result is the execution of lot of database accesses. Database access is an expensive operation. So, end result will be a significant performance trade off. It means an optimum solution should accomplish the task of graph transformation with minimum database accesses.

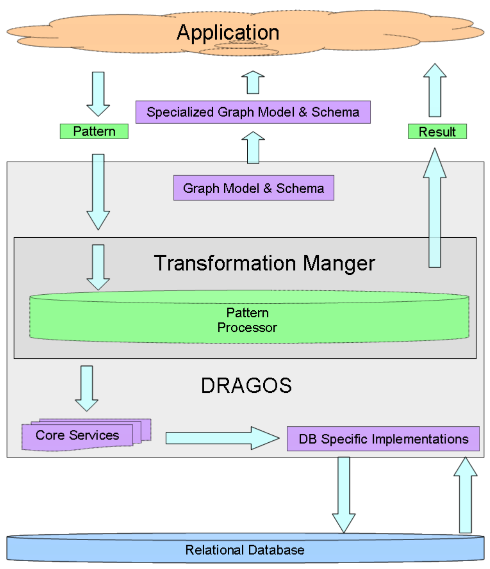


Figure 1.1: QTM without SQL Transformation

1.1.2 Back-end Specific Solution

The second approach is based on direct utilization of DRAGOS‘s support for relational databases as storage back end. It requires development of a transformation mechanism which transforms a pattern in to a SQL statement. Then that SQL statement is executed inside the backend RDBMS. Construction of a single SQL statement for complex pattern reduces the number of database accesses to only one. So, the need of minimization of database accesses which was discussed in universal solution approach is achieved. The figure 1.2 elaborates the back-end specific solution.

The approach very different from universal solution approach. In this, SQL statement is sent directly to database, thus, by passing DRAGOS core services. As already discussed, universal solution on the other hand constructs solution using atomic operations provided by core services which access the DRAGOS‘s database implementation. Execution of this single SQL statement compared to multiple SQL statements to achieve the same solution is a significant performance gain. The disadvantage of this approach is this solution is backend specific and it does not function in in-memory version of DRAGOS.

Modern database systems employ various algorithms and techniques to ensure fast retrieval of data. The execution of SQL statement inside the RDBMS is going to benefit from RDBMS‘s features like query optimization, caching and indexing. The performance gain provided by this approach has also been included in this document in chapter 5.

1.2 Objective of Thesis

The objective of this thesis is to add a graph transformation support in DRAGOS graph database system using its back end RDBMS support. A query and transformation mechanism (QTM) is developed which represents a graph transformation system.

SQL, the data representation, manipulation and query language of RDBMS is used to serve as an implementation technology for a mapping from graph transformation systems to relational databases. QTM process a complex pattern and transforms it in to a single SQL statement. A brief overview of the QTM is given in figure 1.2.

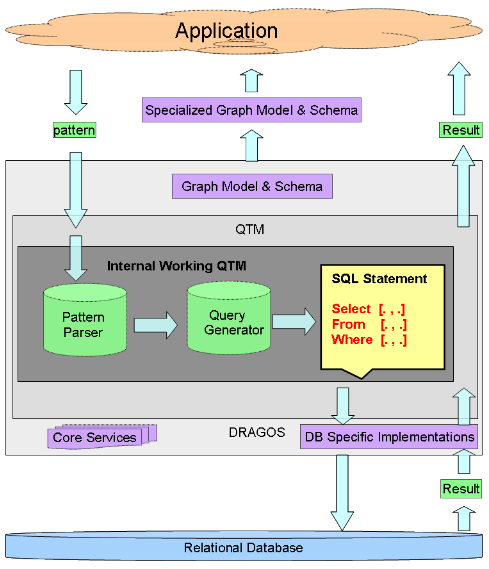


Figure 1.2: QTM: Back-end Specific Solution

QTM comprises of following two parts.

- Pattern Parser
- Query Generator

Pattern Parser, parses the pattern. It accepts the pattern in DRAGULA 3.5. It prepares the pattern in a form readable for query generator.

Query Generator generates a SQL statement (as described in chapter 4) based on the contents of the pattern. As a SQL statement is composed of three parts i.e. select, from and where. Final SQL statement for every pattern is made after making modifications during processing of a pattern depending on its contents. This statement is then executed in a relational database.

Complete Workcylce of QTM

A complete work cycle of the Query and Transformation Mechanism (QTM) shown in figure 1.2 can be explained in following steps.

- Pattern is received by QTM
- Pattern is parsed by the Pattern Parser
- Query Generator generates SQL statement

- Generated SQL statement is then executed in RDBMS. Caching is done of these SQL statements to improve performance.
- DRAGOS gets the result of SQL statement back from RDBMS. This result is the LHS of the graph transformation system. This has been explained in section 3.1.2.

1.3 Example Schema

For better understanding of the document, an example has been added in this section and frequent references to this example will be made during remaining chapters. University Model has been taken as example schema. Its a simplified version of a university administration system that manages the personal and academic information of students and professors. The structural relationships among the classes defined in the schema are given in Figure 1.3.

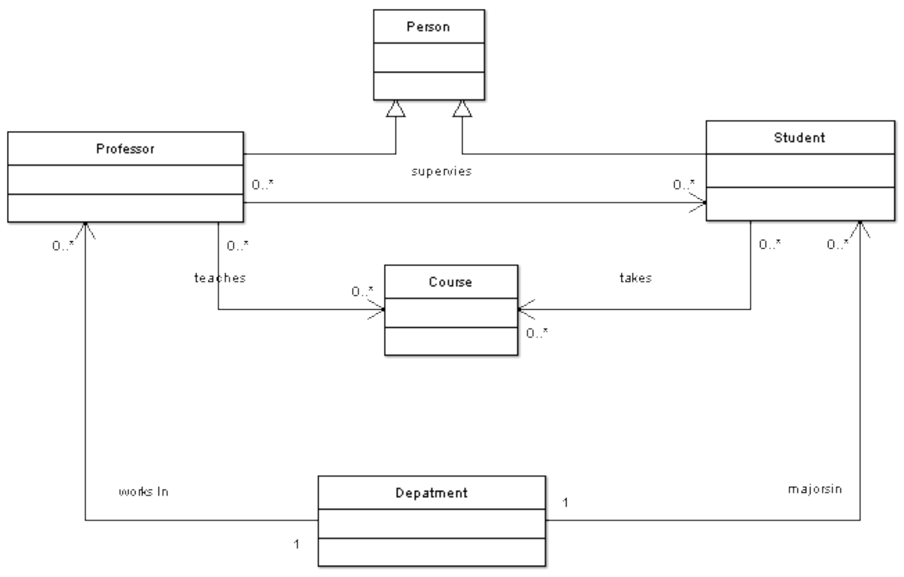


Figure 1.3: Computer Science University Schema

The class Person has two subclasses Professor and Student. A student is supervised by a professor. A professor may be a supervisor for one or more students. However, some students do not have a supervisor and some professors do not supervise a student. Every professor works in a department and every student studies a specific major in a department. Each department offers courses for students. The professor teach these courses.

It means, the schema, when translated to graph technology contains following five nodes.

1. Person
2. Student

3. Professor
4. Course
5. Department

There are five edges in the schema. which are:

1. supervises, relates a professor and a student. A new edge is created when a professor is given a student to supervise.
2. teaches, relates a professor and a course. A new edge is created when a course is taught by a professor
3. takes, relates a student and course. A new edge is created when a student takes a course in the university.
4. employs, relates a department and a professor. A new edge is created when a professor is asked to work for a department.
5. majorsin, relates a student and a department. A new edge is created when student decides to major in a particular department.

1.4 Structure of Thesis

The structure of this document is as follows.

In chapter2, technical background is given to understand QTM. It includes a brief look at DRAGOS with an insight on its graph schema and graph model. it explains, how complex graphs are stored in DRAGOS. It provides overview of relational database management system, concepts of relational database system, relational model, relational operations and database design principles required to understand the QTM. Lastly in this chapter a look at the QTM related tables from the data model of DRAGOS .

Chapter 3 gives the conceptual background necessary to understand QTM. It includes a look at the theoretical background of graph transformations. It has an overview of query transformation language (QTL), its theoretical background and usage. It provides a look at the concept of pattern. Lastly it includes a look at relational algebra and its QTM related background information. This chapter looks at the possible options in DRAGULA for pattern construction.

Chapter 4 provides details on query generation. It explains how SQL is generated for different components of DRAGULA. An example pattern is taken and passed through various phases to generate a complete SQL statement at the end.

In chapter 5, the runtime experiments conducted with QTM are given. It compares efficiency of QTM with implementation of same solution in-memory and PROGRES.

Chapter 6 contains the work related with this thesis. It draws comparison with the existing work done on the topics related to QTM. HQL, UDM and PROGRES have been discussed in this chapter.

Chapter 7 provides the conclusion of this thesis work. It summarizes the learning experiences during the course of this work and directions for any future enhancement of the existing work.

Chapter 2

Technical Background

This chapter provides a look at the technical aspects necessary to understand QTM. This chapter is divided in three sections. With a section on relational databases, DRAGOS and on its database design each. Understanding of these concepts is mandatory for clear understanding of QTM.

As QTM performs graph transformation on relational databases, first concept which require understanding is relational database itself. First section of this chapter is based on broad overview of relational database system and its standard data manipulation language. Then it looks the mathematical foundations of relational model and how equivalent SQL of its mathematical operations is generated.

Section two of this chapter looks at DRAGOS. It provides an overview of the product, its usage and its features. Then it looks at the architecture of DRAGOS. It discusses how, why and where QTM should be added in DRAGOS. It looks at graph schema and graph model of DRAGOS.

Last section of this chapter is based on the database model of DRAGOS. It considers only those database tables which are relevant from QTM's perspective. It shows how few sample insertions can be made in DRAGOS. In last sample data is created for example provided earlier in section 1.3.

2.1 Relational Database System

Database management systems (DBMS) are probably one of the most successful and most widely used products of software engineering. Various types of DBMS systems exist based on the models used (e.g) hierarchal model, network model, relational model, objected oriented model etc. QTM is only concerned with relational databases as DRAGOS uses relational databases as underlying storage.

Relational databases are database systems based on relational model. Most popular commercial and open source databases currently in use are based on this model. DRAGOS currently supports MySQL, PostGreSQL and the Derby database systems. A Relational Database constitutes of set of relations (usually referred as '*tables*'). A relation is a set of tuples ('*records*'), while a tuple is a set of attribute values (usually called '*columns*'), each attribute is identified by its name. The definition of a relational database results in a table of metadata or formal descriptions of the tables, columns, domains, and constraints.

SQL is the standard data manipulation and query language in relational database. Applications access data by specifying queries, which use operations such as select to identify tuples, project to identify attributes and join to combine relations. Relations can be modified using the insert, delete, and update operators. Queries identify tuples for selection, updating or deleting.

Next section summarizes the relational database related terminologies used through out this document. For better understanding an example of a book database system is taken. The database system is required to store ISBN number, book title and author name of a book. the system should be able to store a customer's order containing customer name and the books it ordered. Suppose there are two books to be stored. One is the autobiography of Nelson Mandela name 'Long Walk to Freedom' with ISBN number 0316548189. Second book is Mohandas Gandhi's autobiography called 'The Story of My Experiments with Truth' with ISBN number 0486245934. One customer Smith buys Ghandi's book and another named Peter has bought Mandela's book. This example will be used through out this section.

2.1.1 Relation

The term relation is referred as *'table'* in common RDBMS terminologies. A table is the basic entity of a relational database. A relation is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical real life concepts. All the data referenced by an attribute belong to the same domain and conform to the same constraints. A database table with n columns denoted by $\tau(A_1, \ldots, A_n)$is a relation. τ and A_i denote the name of the table and of the ith column respectively. Column names have to be unique in the scope of the a single table, thus a table cannot have columns sharing the same name. Elements of a table are called tuples and are called rows in database terminology. Similarly, The attributes are columns in database terminology.

Book database example given in Section 2.1 has two concepts namely book and order. So two tables are made namely *'Book'* as shown in table 2.1 and *'Order'* as depicted in table 2.2. The example required storage of three attributes of book namely ISBN number, title and author name. The *'Book'* table has three columns as well. There are two books to store in database, so, the table books contains two rows. System required customer name and books ordered to be stored for every order. The table *'Order'* has two columns. Two orders were placed in example, *'Order'* tables contains these rows. Two more database concepts related to relation that require understanding before proceeding forward are primary key and foreign key. They are discussed in the next sections.

ISBN	Title	AuthorName
0316548189	Long Walk to Freedom	Nelson Mandela
0486245934	The Story of My Experiments with Truth	Mohandas Gandhi

Table 2.1: Table Book

CustomerName	ISBN
Smith	0486245934
Peter	0316548189

Table 2.2: Table Order

Primary Key

A primary key constraint for columns $A_1, A_2, \ldots, A_j$ of table $\tau(A_1, \ldots, A_n)$ guarantees the uniqueness of values in the selected column. Formally, $\forall r, s \in \tau : (r = s \iff \forall i, 1 \leq i \geq j$: $r[A_i] = r[A_j])$ [41]. In other words in relational database, a primary key is a unique key to identify each row in a table. The International Standard Book Number (ISBN) is a 10-digit number that uniquely identifies books and book-like products published internationally. In table 2.1 ISBN is the primary key as it is unique for every row. There can never be two books with same ISBN number.

Foreign Key

For two relations $\mathcal{R}$ and $\mathcal{S}$, a foreign key constraint for column $\mathcal{R}$.A referring to Column $\mathcal{S}$.B (denoted by $\mathcal{R}.A \xrightarrow{FK} \mathcal{S}.B$) declares that all values of column R.A should also be found in column R.B or formally R.A$\sqsubseteq$$R$.B [41]. In other words in relational databases, a foreign key is a referential constraint between two tables. The foreign key identifies a column or a set of columns in one (referencing) table that refers to a column or set of columns in another (referenced) table. The columns in the referencing table must be the primary key in the referenced table[39]. The values in one row of the referencing columns must occur in a single row in the referenced table. In table 2.2 ISBN number is the foreign key.

2.1.2 Standard Query Language

As discussed earlier, SQL is a standardized language for defining and manipulating data in a relational database. In accordance with the relational model of data, the database is treated as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more base tables [23].

SQL is built upon precise mathematical foundations. SQL statements are executed internally by the a database system employing various modern techniques for query optimization. SQL is a standard interactive and programming language for getting information from and updating a database. Although SQL is both an ANSI[19] and an ISO standard[16], many database products support SQL with proprietary extensions to the standard language. As DRAGOS supports various relational databases, this issue of proprietary extensions by different is handled in DRAGOS by using *Template Based Code Generation*[28] which will be defined later in this document. Queries take the form of a command language that makes selection, insertion and updation possible. The table2.3 summarizes some of the most important relational database terms and their SQL database equivalents.

As already discussed relational databases are based on relational model. It was proposed by Dr. Codd in 1969. It contains relational operations and

Relational Term	SQL equivalent
relation (base relation)	table
relation (derived relation)	view, query result
tuple	row
attribute	column

Table 2.3: Relational SQL Equivlant

relational operators. Next section describes this model and equivalent SQL of its operators and operations.

2.1.3 Relational Algebra

Relational Algebra (RA) is the lower level data manipulation language for relational model. It consists of several basic operations which is enable user to specify retrieval requests. In his original relational algebra, Codd introduced eight relational operations and operators in two groups of four each respectively. The first four operators were based on the traditional mathematical set operations, which are:

1. Union
2. Intersection
3. Difference
4. Cartesian product

Before proceeding further, and looking at all of these operations one by one, there is a need to define, when are two relations union compatible. This is required because Union, Intersection and Difference operations can only be performed on *Union Compatible Relations.*

Union Compatible Relations

Two relations r(A1, A2, . . . , An) and s(B1, B2, . . . , Bn) are union compatible if they have the same degree n and $dom(Ai) = dom(Bi) for 1 \leq i \leq n$ [38].

This means two union compatible relations must have the same number of attributes and each corresponding pair of attributes have the same domain. In terms of SQL both relations must have same number of column with identical data types.

Union

The Union operator combines two *Union Compatible Relations* into a single relation via set theory union operation between two tuple sets. For two relations $r1$ and $r2$, who are union compatible in a database schema R. Union between two relations is written as:

$$Notation : r1 \cup r2$$

Formally union can be described as:
$r1 \cup r2 = \{\ t \mid t \in r1 \vee t \in r2\ \}$ where $r1(R) = r2(R)$

A Union operation for given relations $r1$ and $r2$ can be performed in the following steps:

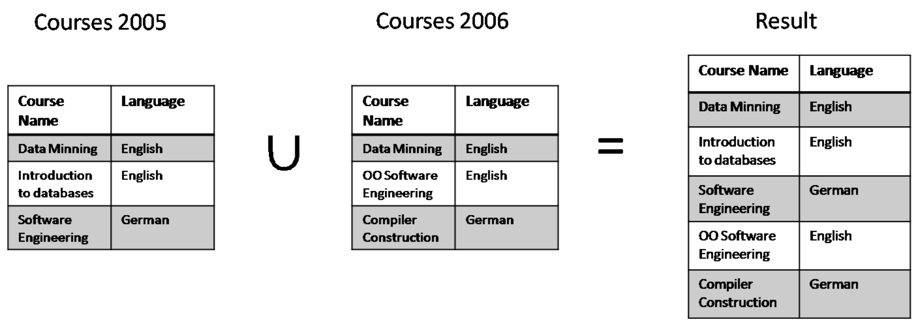
- Make copy of first relation r1
- For each tuple t in relation r2, check whether it is in the result or not. If it is not already in the result then place it there.

Union Operator in SQL

$\cup$ (Union) operator in relational algebra is equivalent to *UNION* operator of SQL. In SQL the *UNION* operator combines the results of two SQL queries into a single table for all matching rows. The two queries must have the same number of column references and identical data types of corresponding columns to unite. Any duplicate records are automatically removed.

Example The tables in 2.1 and 2.2 are not union compatible as these two tables do not have same number of columns. A simple example to show union would be a database having tables courses2005 and courses2006 as showed in figure 2.1. They have identical structures but are separated because of performance considerations. A *UNION* query would combine results from both tables. The visual description of *UNION* operation is provided in figure 2.1. The equivalent SQL statement is given as under:

```
Select * From courses2005
Union
Select * From courses2006
```



Courses 2005

Course Name	Language
Data Minning	English
Introduction to databases	English
Software Engineering	German

$\cup$

Courses 2006

Course Name	Language
Data Minning	English
OO Software Engineering	English
Compiler Construction	German

=

Result

Course Name	Language
Data Minning	English
Introduction to databases	English
Software Engineering	German
OO Software Engineering	English
Compiler Construction	German

Figure 2.1: Union Operation Example

Intersection

The INTERSECTION operator combines two union compatible relations into a single relation set theory intersection operation between two tuple sets. For two relations r1 and r2, who are union compatible in a database schema R. Intersection between two relations is written as:

$$Notation : r1 \cap r2$$

Formally intersection can be defined as:
$r1 \cap r2 = \{\ t \mid t \in r1 \wedge t \in r2\ \}$ where $r1(R) = r2(R)$

A Intersection operation for given r1 and r2 can be performed in following steps:

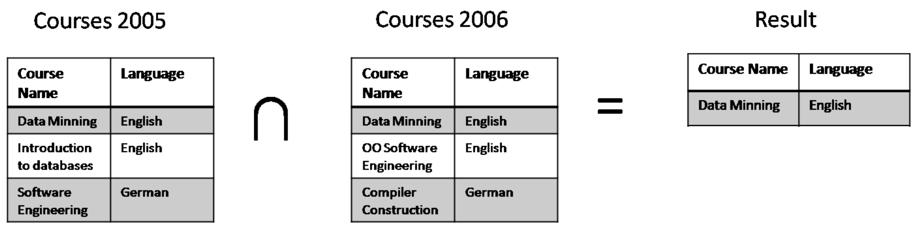
- Initially, result set is empty
- For each tuple t in relation r1, if t is in the relation r2 then place t in the result set.

Intersection Operator in SQL

$\cap$ (Intersection) operator of relational algebra is equivalent to SQL's *INTERSECT* operator. The intersection operator produces a single set, which is comprised of those tuples who are common between the two relations. The SQL INTERSECT operator takes the results of two queries and returns only rows that appear in both result sets.

Example To show union same database is taken as in section 2.1.3. A *INTERSECT* query gives a single relation which contains the common elements between the two relations. The visual description of Intersection is provided in figure 2.2. The equivalent SQL statement is given as under:

```
Select * FROM courses2005
INTERSECT
Select * FROM courses2006
```



Courses 2005

Course Name	Language
Data Minning	English
Introduction to databases	English
Software Engineering	German

$\cap$

Courses 2006

Course Name	Language
Data Minning	English
OO Software Engineering	English
Compiler Construction	German

=

Result

Course Name	Language
Data Minning	English

Figure 2.2: Intersection Operation Example

Difference

The Difference operation finds the set of tuples that exist in one relation but do not occur in the other *Union Compatible Relation*. For two relations r1 and r2 who are union compatible in a database schema R. Difference between two relations is written as:

$$Notation : r1 \backslash r2$$

Formally difference can be defined as:
$r1 \backslash r2 = \{\ t \mid t \in r1 \wedge t \notin r2\ \}$ where $r1(R) = r2(R)$

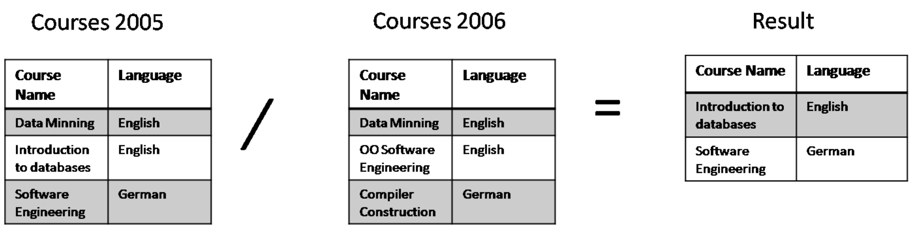
- Initially, result set is empty
- For each tuple in r1, check whether it appear in r2 or not. If it does not then place it in the result set. Otherwise, ignore it

Difference Operator in SQL

SQL's equivalent operator of \ operator of relational algebra is *EXCEPT*. The difference operator acts on two relations and produces a single set, which is comprised of tuples from the first relation that do not exist in the second relation.

Example To show union same database is taken as in section 2.1.3. A *EXCEPT* query gives a single relation which contains only the elements which are contained in first relation but not in second relation. The visual description of Difference is provided in figure 2.3. The equivalent SQL statement is given as under:

```
Select * FROM courses2005
EXCEPT
Select * FROM courses2006
```



Courses 2005

Course Name	Language
Data Minning	English
Introduction to databases	English
Software Engineering	German

/

Courses 2006

Course Name	Language
Data Minning	English
OO Software Engineering	English
Compiler Construction	German

=

Result

Course Name	Language
Introduction to databases	English
Software Engineering	German

Figure 2.3: Relational Algebra Difference Operation

Cartesian Product

The Chartesian Product operation combines information from two relations pairwise on tuples. For two relations r and s in database schema R. Chartesian Product between two relations is written as:

$$Notation : r \times s$$

Formally chartesian product can be described as:

$r \times s = \{(t1, t2) \mid t1 \in r \wedge t2 \in s\}$ where $r(R) = s(R)$

Chartesian product operation between the given relations r and s can be performed in following steps:

- For each tuple in r, form new tuples by pairing it with each tuple in s
- Place all of these new tuples in the result set

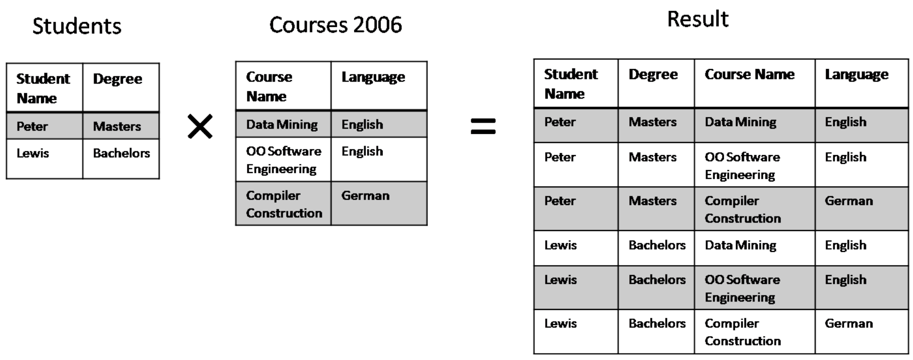
Chartesian Product Operator in SQL

The cartesian product of two relations is a join that is not restricted by any criteria, resulting in every tuple of the first relation being matched with every tuple of the second relation. The cartesian product is implemented in SQL as the *CROSS JOIN* operator.

Example

A simple example to show chartisean product would be a database having tables students and courses2006 as showed in figure 2.1. A *CROSS JOIN* query would combine results in form of a chartesian product. The visual description of *CROSS JOIN* operation is provided in figure 2.4. The equivalent SQL statement is given as under:

```
Select * FROM
courses2006
        CROSS JOIN
Students
```



Students

Student Name	Degree
Peter	Masters
Lewis	Bachelors

×

Courses 2006

Course Name	Language
Data Mining	English
OO Software Engineering	English
Compiler Construction	German

=

Result

Student Name	Degree	Course Name	Language
Peter	Masters	Data Mining	English
Peter	Masters	OO Software Engineering	English
Peter	Masters	Compiler Construction	German
Lewis	Bachelors	Data Mining	English
Lewis	Bachelors	OO Software Engineering	English
Lewis	Bachelors	Compiler Construction	German

Figure 2.4: Relational Algebra Product Operation

The remaining operations proposed by Codd are as follows:

1. Selection
2. Projection
3. Join operation
4. Relational division

The next sections discuss all these operations one by one.

Selection

The selection operation has one relation as input and output respectively. The output relation is a subset of input relation containing those tuples who satisfy the given selection condition. Basically it partitions the input relations in to two sets of tuples (i) those tuples that satisfy the condition are selected and (ii) those who do not satisfy the condition are discarded. For a relation r in a database Schema R the selection operation is written as follows:

$$Notation : \sigma_{selection-condition}(r)$$

Formally selection can be defined as:
$\sigma_F(r) = \{t \mid t \in r \wedge F(t)\}$ where F is a boolean expression on attributes in r. The selection condition is made up of a number of clauses of the form

- <attribute name> <comparison op> <constant value> OR/AND

- <attribute name 1> <comparison op> <attribute name 2>

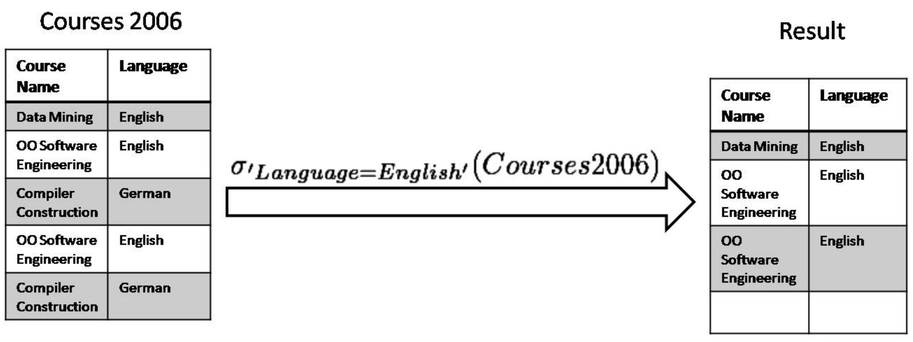
The clauses are connected by boolean operators 'AND' and 'OR' operators. In the clause, the comparison operations could be one of the following $\leq$, $\geq$, $\neq$, $=$, $>$ and $<$.

Selection Operation in SQL

The SQL equivalent of selection operation of relational algebra is the *SELECT* query statement with a *WHERE* clause. The condition is given in the *WHERE* clause.

Example A simple example to show selection would be a database having table courses2006 as showed in figure 2.5. If courses whose language of instruction is English are to be retrieved the visual description of *Selection* operation is provided in figure 2.5. The equivalent SQL statement is given as under:

```
Select * FROM
courses2006
      WHERE
Language = "English"
```



Course Name	Language
Data Mining	English
OO Software Engineering	English
Compiler Construction	German
OO Software Engineering	English
Compiler Construction	German

Course Name	Language
Data Mining	English
OO Software Engineering	English
OO Software Engineering	English

Figure 2.5: Selection Operation in Relational Algebra

Projection

The project operation is another unary operation. This operation returns a set of tuples containing a subset of the attributes in the original relation. Thus, it can be stated that the Selection operation selects some rows and discards the others. The project operation, on the other hand, selects some columns of the relation and discards the other column. The project operation can be viewed as the vertical filter of the relation. For a relation r of a database schema R, project is defined as

$$\pi_{attribute-list}(r)$$

Formally for a relation r and, X, a set of attributes of r, projection is defined as follows:

$$\pi_X(r) = \{t[X] | t \in r\}$$

Projection Operation in SQL

The projection operation of relation algebra in SQL is also done by *SELECT* operator. One difference between projection and selection is that in projection after *SELECT* operator, the names of attributes are provided. Projection does not care about duplicates. In order to remove duplicates *DISTINCT* keyword is used in SQL.

Example: A simple example to show projection would be a database having table courses2006 as showed in figure 2.6. If only the column coursename is to be retrieved. The visual description of this *Projection* operation is provided in figure 2.6. The equivalent SQL statement is given as under:

```
Select coursename FROM
courses2006
```

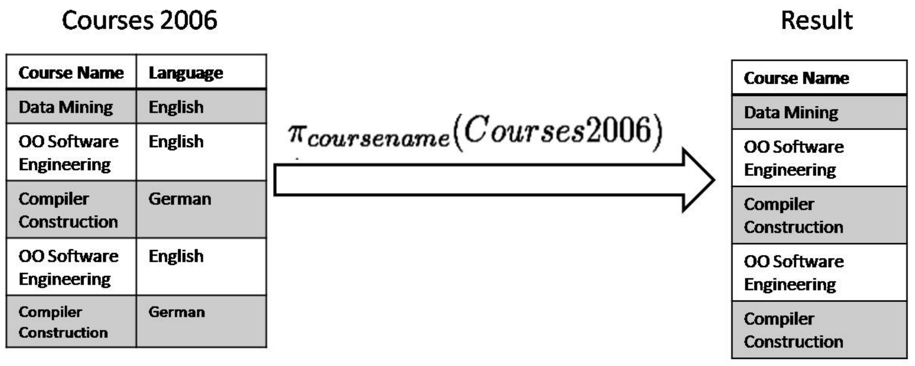


Figure 2.6: Projection Operation in Relational Algebra

JOIN Operation

The join operation is used to combine related tuples from two relations into a single tuple. There are various kinds of joins such as natural join, equi join, outer joins. From QTM's perspective only natural join is relevant. So, here only natural join is discussed. For two relations r and s in a database schema R, natural join is described as:

$$Notation : r \bowtie s$$

Formally natural join can be described as:

$$r \bowtie s = \pi_{r \cup s} \sigma_(r \times s)$$

Natural join operation for given relations r and s can be performed in following steps:

- For each tuple in relation r, compare common attributes with those in each tuple of s
- If two tuples match in their common attributes then combine tuples, remove duplicate attributes and add to the result.

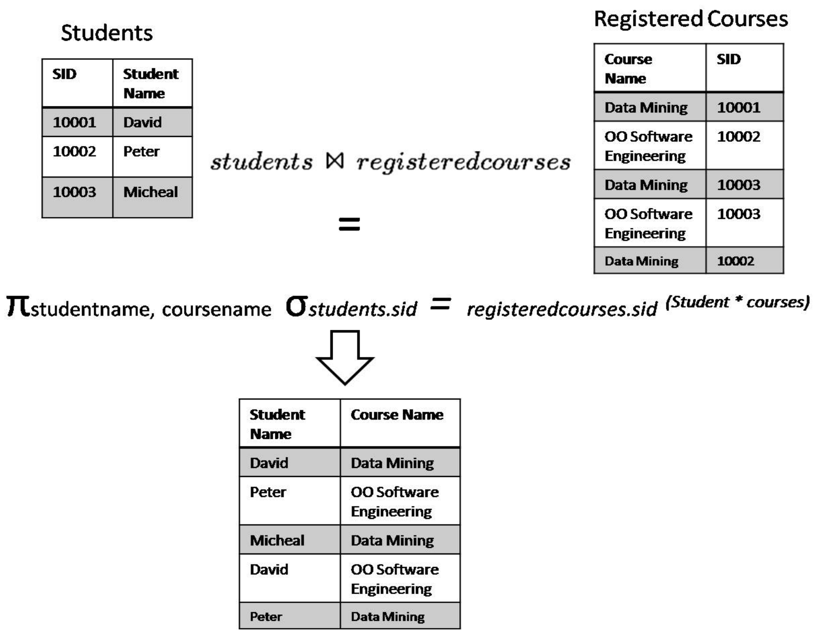
Join Operation in SQL

The join operation of relation algebra in SQL, like projection and selection operations is also done by *SELECT* operator. The join operations combines two relations over a join condition. The resultant relation contains only those tuples that satisfy this condition.

Example:

A simple example to show join operation would be a database having tables registeredcourses and students as showed in figure 2.6. Table student contains all students and registeredCourses contains courses registered by those students. The coursename column of registeredcourses contains the coursenames and SID column contains the id of the student who has registered that course. The visual description of this *Projection* operation is provided in figure 2.6. The equivalent SQL statement is given as under:

```
Select studentname , coursename FROM
courses2006 , students
WHERE
students . sid = courses2006 . sid
```



Students
Registered Courses
students ⋈ registeredcourses
=
πstudentname, coursename σstudents.sid = registeredcourses.sid (Student * courses)

SID	Student Name
10001	David
10002	Peter
10003	Micheal

Course Name	SID
Data Mining	10001
OO Software Engineering	10002
Data Mining	10003
OO Software Engineering	10003
Data Mining	10002

Student Name	Course Name
David	Data Mining
Peter	OO Software Engineering
Micheal	Data Mining
David	OO Software Engineering
Peter	Data Mining

Figure 2.7: Join Operation in Relational Algebra

Division Operation

The division is a binary operation which requires two relations. The relational division operation is slightly more complex operation, which involves essentially using the tuples of one relation (the dividend) to partition a second relation (the divisor). The relational division operator is effectively the opposite of the cartesian product operator. For two relations r and s in a database schema R, division can be written as:

$$Notation : r \div s$$

2.1.4 Database design

In software engineering the term database design is the process of producing detailed data model of a database based on the application requirements. This logical data model contains all the needed logical and physical design choices and physical storage parameters needed to generate a design in a Data Definition Language, which can then be used to create a database. A fully attributed data model contains detailed attributes for each entity. A data model is an abstract model that describes how data is represented and accessed. In software industry, the term datamodel is often a reference to a document which contains the logical and physical structure of an underlying database of an application.

2.2 DRAGOS Graph Database

DRAGOS (Database Repository for Applications using Graph Oriented Storage), as discussed earlier is a graph database system. It can store and retrieve complex graph structures. Using graphs as as a fundamental data model ensures that even complex data structures are stored easily without the need of helper elements. Its unlike relational model where additional storage elements are required to cater many-to-many relations.

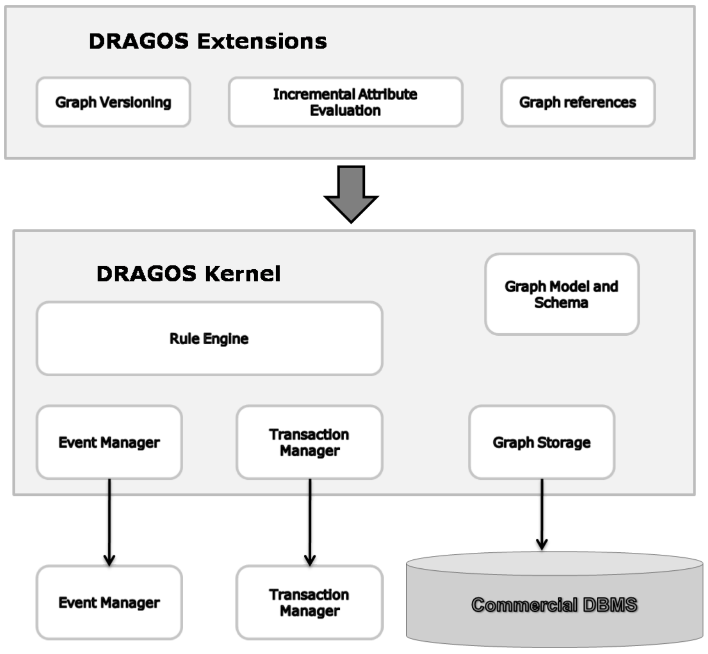


Figure 2.8: DRAGOS Architecture

DRAGOS can be used by software applications such as Integrated development environment (IDE), computer aided software engineering tools(CASE), reverse engineering tools and other interactive applications who use complex object structure. The common feature of all these applications from various application domains is the necessity to handle different types of objects at different levels of efficiency. All use coarse and fine grained objects which can have complex hierarchical or non hierarchal inter references. These applications usually require vast number of differently sized attributes. DRAGOS provides solution to all of the above mentioned demands of applications. It effective handling of complex graph objects, their inter-referencing and it supports attribute storage for graph objects.

DRAGOS has a multilayer architecture. It is implemented in java which makes platform independent. An overview of the structure of DRAGOS is provided in Figure 2.8. Next section provides an overview of the structure of DRAGOS.

Structure of DRAGOS

DRAGOS is structured in to a multi-layered architecture which gives it significant advantages specially in the maintenance and evolution stage. As technologies and functionality requirements change over time, this layered architecture makes adaptation easier for DRAGOS. Each layer of DRAGOS includes different components. DRAGOS includes components for graph versioning, attribute evaluation etc. The structure of DRAGOS system is illustrated in Figure 2.8. Next sections look at these layers one by one.

2.2.1 DRAGOS Backend Storage

DRAGOS does not implement a dedicated storage module for graph storage, instead it uses existing relational database systems for graph storage. DRAGOS internally implements a mapping mechanism between graphs and RDBMS for storing graphs. It supports many relational database systems such as MySQL[30], PostgreSQL [26] and Derby [21] etc. DRAGOS has for every database, JDBC[22] (java Database Connectivity) based written modules available, each of which has its own graph model implementation. The DRAGOS user only has to specify the database to be used, the rest is done by that implementation itself. For testing purposes a fast in-memory model is available.

2.2.2 DRAGOS Kernel Layer

DRAGOS Kernel is the most significant layer of DRAGOS and provides most of the services. Services provided by Kernel include opening and closing of database connections, graph storage management, transaction management, event management, and graph schema creation.

Graph Schema defines the fundamental structure of graph and graph model is its instance. The next section describes graph model and details about graph schema are included in section 2.2.2.

Graph Model

DRAGOS offers a rich graph model originally inspired by the Graph eXchange Language (GXL) [18]. It is an XML based exchange format, which is applied for sharing data between software re-engineering tools. The major aim of the GXL is to provide interoperability between tools which utilize graphs for the representation of internal data. Typed, attributed and ordered directed graphs constitute the base for GXL. In DRAGOS like specified by GXL nodes, edges and relations can be identified and attributed [44]. Graph model is an instantiation of graph schema.

The DRAGOS graph model is illustrated as a UML class diagram in Figure 2.9. Graph model instantiates nodes, edges and relations between these nodes and edges. The instantiation is done of corresponding nodes, edges declared in schema. Graph model contains classes for every graph element. All these classes as shown in the graph model class diagram in 2.9 are inherited from a GraphEntity class.

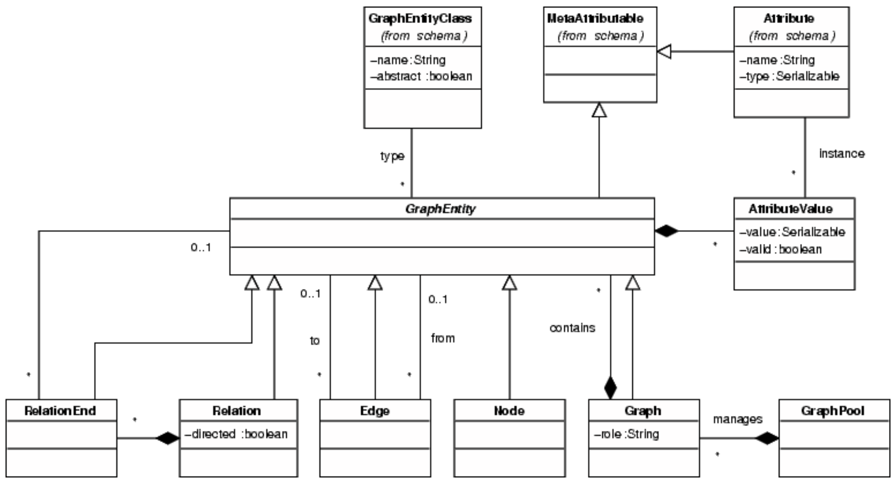


Figure 2.9: Graph Model [13]

All graphs in the database are managed by a graph pool as shown graph model class diagram. GraphPool object represents graph pool. In the graph pool, an arbitrary number of graphs can be stored. These graphs are identified by their name, which must be unique. A graph pool is capable of storing any number of graphs but it contains only one graph schema.

A graph contains an arbitrary number of graph entities such as edges, nodes, relations, and relation ends. which are all inherited from GraphEntity. A graph is also one of the possible graph entities. Every graph entity can be attributed in DRAGOS. The values of the attributes are represented in the Dragos graph model by the class AttributeValue.

The graph object instantiates nodes, edges and relations. Object of class Node is given on creation of a node, object of Edge is given for every edge and object of class Relation is given on creation of every relation. However, the relation ends are not directly part of graphs. Instead, a relation end belongs to relation. As already discussed, every graph entity can be attributed in DRAGOS. The values of the attributes are represented in the DRAGOS graph model by the class AttributeValue. The difference between edges and relations is that the relations can be connected with any number of graph entities by the relation ends.

Graph Schema

Graph schema specifies the contained graph elements, there types and attributes associated with them. It also specifies the relations between these graph elements, cardinalities of these relations and lastly the constraints over relations ships.

The DRAGOS schema is represented in Figure 2.10 as a class diagram. For storage of a graph, first step is the creation of its schema. Graph schema defines what nodes and edges exist in a graph and which relations can exist between these nodes and edges. DRAGOS schema declares a class for creation of every graph element (e.g) NodeClass for a node, EdgeClass for an edge etc. All of these classes as shown in the schema class diagram in 2.10 are inherited from a GraphEntityClass.

GraphEntityClass is abstract, so, it cannot be instantiated. Each declared GraphEntityClass is required to identify itself with a unique name in string format. This unique name acts as an identifier for every GraphEntityClass. The GraphEntity class object can later be retrieved from schema using this name. An arbitrary number of attributes can be defined for every GraphEntityClass. Every attribute, like other GraphEntityClass objects is identified by a name that has to be unique for each graph entity class. The attributes inheritance from super classes is not supported. Every attribute also has a type that defines which values are valid.

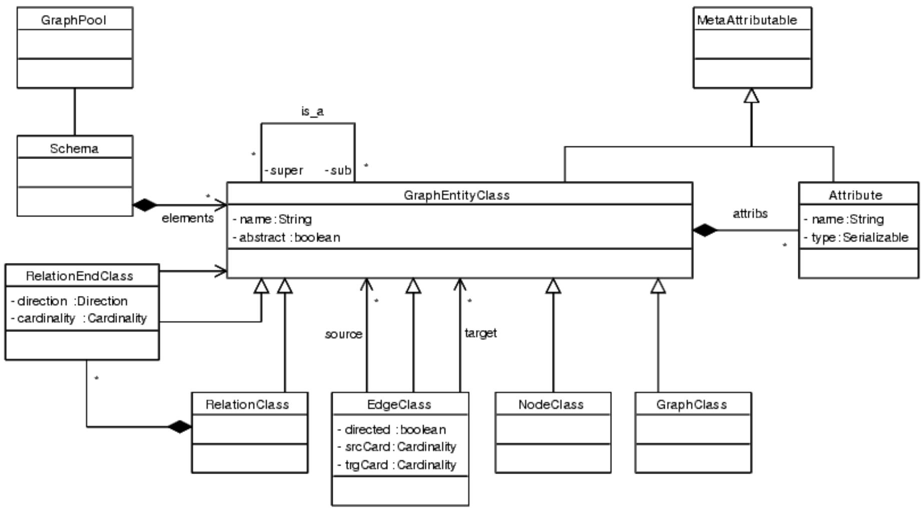


Figure 2.10: Graph Schema [13]

After describing the superclass GraphEntityClass, graph element classes are discussed one by one. As already described graph schema declares a GraphEntityClass for every graph element in a graph. It contains NodeClass for creation of every node, EdgeClass for Edge, GraphClass for graph and, RelationClass and RelationEndClass for relations.

For every node in a graph is instantiated from a NodeClass. Every new NodeClass requires a unique name in string. The unique string name is then used to retrieve this NodeClass object from storage. This NodeClass can declares its attributes.

An edge connects two nodes. So, for creation of an edge class, DRAGOS schema requires information about the source, the target and the cardinality of there relationship. For this, the cardinality attribute is specified on the source and target nodes. The DRAGOS graph schema supports directed as well as undirected edges. This is realized using the attribute called directed. The inheritance hierarchy is also considered. It means subclasses of an entity class are also permitted.

Relations can be connected to any number of graph entities. For the realization of this feature, the RelationClass and the RelationEndClass are defined. For the declared RelationEndClass classes, it is defined, which cardinality the RelationEndClass may have and which GraphEntityClass are allowed as target and whether the relation ends are directed or ordered. A name is specified for every RelationEndClass. The relation end classes are related to the relation classes. Thereby which relation ends may be connected by a relation is specified.

2.3 Datamodel of DRAGOS

As already discussed in section 2.1.4, in software engineering the term datamodel is often a reference to a document which contains the logical and physical structure of an underlying database of an application. DRAGOS stores graph structures. It has various storage options but from QTM's perspective only relational database is relevant. QTM generates a single SQL statement for every pattern. To understand the process of generation of SQL statement, understanding of underlying data model is necessary. Due to this reason overview of data model of DRAGOS is provided.

DRAGOS has a complex data model which includes numerous tables. This section only looks at tables which are relevant to the QTM. Nodes, edges and relations are the graph elements which constitute a graph. QTM performs queries over these graph structures. So, its important to understand how nodes, relations and edges are stored (i.e) which tables are required. This section includes sample insertion of nodes and edges taken from example provided in Section 1.3.

The table 'GraphEntityClass' is the first one that QTM uses. As already described in DRAGOS graph schema 2.2.2, all graph elements are instantiated from a 'GraphEntityClass' object. This object is represented in database by table 'GraphEntityClass' as shown in table 2.4. This table has four fields. There description and data types are as follows:

1. **Id**: A unique identifier for every graph entity class. Its an integer which is incremented on every new insertion of a graphentityclass. It the primary key of the table.

2. **Type**: 'Type' field describes whether the stored graph entity class is a a node, edge or graph. It is an integer. DRAGOS uses constants to represent edge class and node class (i.e) constant for edge class is 202, for graph class it is 206 and a node has 204.

3. **name**: Every graph entity class entry requires a unique name. This attribute is of data type string.

4. **abstract**: It indicated whether the graph entityclass is abstract or not. It is of data type boolean.

Graph Entity Class			
id	type	name	abstract

Table 2.4: Table Graph Entity Class

DRAGOS's support for different RDBMS system is an attractive feature. Up to now, it supports MySQL[30], PostgreSQL [26] and Derby [21]. However, the datamodel is not exactly the same for all these databases. Datamodel for PostgreSQL is different from rest of the database systems. So the datamodel of DRAGOS can be divided in two categories. Which are:

1. General Datamodel

2. Database specific DataModel

2.3.1 General Datamodel

So far, apart from PostgreSQL all the database systems supported by DRAGOS have identical datamodel. The general datamodel requires insertion of node only in table 'GraphEntity' as shown in table 2.5. Edge who is also a graph entity, is stored in tables 'GraphEntity' shown in table 2.5 and 'EdgeData' as shown in Table 2.6. The 'id' field of 'GraphEntity' table acts as the foreign key in table 'EdgeData'.

Graph Entity			
id	type	class	parent

Table 2.5: Table Graph Entity

Table GraphEntity as depicted in Table 2.5 has four fields. There description and data types are as follows:

1. **Id**: It is unique for every graph entity. Its an integer which is incremented on every new insertion of a graphentity. It the primary key of the table.

2. **Type**: It is derived from this field that whether the stored graph entity is a a node, edge or graph. It is an integer. DRAGOS has constants to represent edges and nodes (i.e) value for constant for edge is 22, for graph it is 26 and for node it is 24.

3. **Class**: As every graph element is instantiated from a 'GraphEntityClass'. It identifies the GraphEntityClass thie graph entity belongs to. It is the foreign key and contains the value given in id field of the GraphEntityClass table. It is of data type integer.

4. **Parent**: It is id of the parent graph of the current graph entity. In case of a graph its null. It is an integer.

The Edge data looks like this.

Edge Data		
id	source	target

Table 2.6: Table Edge Data

Table EdgeData as shown in Table 2.6 contains three fields. There description and data types are as follows:

1. **Id**: It is the foreign key linked with table GraphEntity. It points to the graph entity this edge belongs to. It is of data type Integer.

2. **Source**: It contains the id of the source of the edge. It is an integer. Source is the unique identifier i.e value of 'Id' field of the graph entity who is the source of this edge.

3. **Target**: It contains the id of the target of the edge. It is also an integer. Target is the unique identifier i.e value of 'Id' field of the graph entity who is the source of this edge.

To access a complete edge object, the tables 'EdgeData' and 'GraphEntity' are required to be joined explicitly on field 'Id'. To provide a clear understanding of the datamodel of DRAGOS, a few sample insertions are provided in following sections using the example given in section 1.3.

2.3.2 Database specific DataModel

In database specific datamodels the example of PostGRSQL is taken. PostgreSQL supports inheritance among tables. It means a table can inherit attributes and fields from another table and relationship among the tables is done internally by DBMS on the basis of a common primary key. As already discussed in section DRAGOS schema 2.2.2, every node and edge is a graph entity too. Datamodel contains one table for storing a graph node named 'GraphEntity'. Edge has a source and a target, and edge is also a graph entity. PostgreSQL has two tables for storing an edge. These are tables 'GraphEntity as depicted in Table 2.7 and table Edge as shown in Table 2.8. The table Edge inherits from table GraphEntity. The design of table 'GraphEntity' is same for PostgreSQL and general datamodel as shown by tables 2.7 and 2.8 respectively. So, there is no need of repeating the field descriptions here. The description earlier given in section on description of general data model is valid for table 'GraphEntity' of the database specific data model.

Graph Entity			
id	type	class	parent

Table 2.7: Table Graph Entity

Edge	
source	target

Table 2.8: Table Edge in PostgreSQL

PostgreSQL supports inheritance, table Edge as show in 2.8 inherits from table graph entity, shown in Table ch3:tabgraphentityPost. It means any SQL statement referencing the table edge can also reference fields in table 'GraphEntity'. The table 'Edge' contains only two fields. There description and data types are as follows:

1. **Source**: It contains the id of the source of the edge. It is of data type integer.

2. **Target**: It contains the id of the target of the edge. It is of data type integer.

2.3.3 Sample Data

As already described in previous Section 2.3 there are two types of relational datamodels of DRAGOS, one for PostgreSQL and one for all others. This section creates a sample data and then explains how insertions of that sample is done in DRAGOS. The insertions made here will be referenced in later chapters as well. Few patterns will be executed on this data and SQL queries will be generated on this data.

Graph Entity Class			
id	type	name	abstract
1998	206	UniversityGraphClass	FALSE
1999	204	Person	TRUE
2000	204	Student	FALSE
2001	204	Professor	FALSE
2002	204	Course	FALSE
2003	204	Department	FALSE
2004	202	takes	FALSE
2005	202	majorsin	FALSE
2006	202	supervises	FALSE
2007	202	teaches	FALSE
2008	202	worksfor	FALSE

Table 2.9: Table Graph Entity Class

Let the university schema of example provided in Section 1.3 has three students namely Thomas, Micheal and Susain. two professors are teaching in university namely Mr. Coenen and Mrs. Steinmayer. The table 'GraphEntityClass' as a result of schema creation is shown in table 2.9. The constant 26 used in field type of table 'GraphEntityClass' denotes a graphclass, similarly constant 22 and 24 represent that rows entries are of type nodeclass and edge class respectively. The entries in table marked blue are edges, green are nodes and the entry in red indicates a graph.

Graph Entity			
id	type	class	parent
33402	26	1998	
33403	24	2000	33402
33404	24	2000	33402
33405	24	2000	33402
33406	24	2001	33402
33407	24	2001	33402
33408	24	2003	33402
33409	24	2002	33402
33410	24	2002	33402
33411	22	2008	33402
33412	22	2008	33402
33413	22	2007	33402
33414	22	2007	33402
33415	22	2004	33402
33416	22	2004	33402
33417	22	2004	33402
33418	22	2004	33402

Table 2.10: Table Graph Entity

The university has a department named Information Systems. Mr. Coenen and Mrs Steinmayar are working for this department. Mr. Coenen is teaching Introduction to databases and Mrs. Steinmayar is teaching implementation of databases. Thomas and Micheal have taken Introduction to databases. Susain has taken both Introduction to databases and implementation of Databases.

To realise the example taken in section 1.3 and its sample data given above, firstly creation of a new graph is required. DRAGOS inserts a new graph in table 'GraphEntity'. When University graph was inserted in DRAGOS, it made entries shown in row one of table 2.10 with id field equal to 33402. On insertion of eight nodes nodes which were three students, two professors, two courses and one department respectively. Eight entries were made in table 'GraphEntity. The rows of table 2.10 with type attribute equal to 24 depicts the node entries. As already discussed, tables which store edges are different for general model and database specific models. The sample data contains eight edges. All rows in table 2.10 with type attribute equal to 22 are edges. However as PostgreSQL supports inheritance, it does not require storage of a foreign key. Data inserted in PostgreSQL is shown in table 2.11. The general model require storage of

Edge	
source	target
33409	33406
33410	33407
33408	33406
33408	33407
33403	33409
33404	33409
33405	33409
33405	33410

Table 2.11: Table EDGE PostGRe

foreign key. So, the graph entity id is stored as foreign key in table EdgeData in the general model as shown in table 2.12. The entries in table marked blue are edges, green are nodes and the entry in red indicates a graph.

Edge Data		
id	source	target
33411	33409	33406
33412	33410	33407
33413	33408	33406
33414	33408	33407
33415	33403	33409
33416	33404	33409
33417	33405	33409
33418	33405	33410

Table 2.12: Table EDGE Data

In chapter 4 various phases of SQL generation are described. It describes steps involved in SQL statement generation. The understanding of the data model of DRAGOS provided here is necessary to understand QTM's most critical process of SQL statement generation. But, understanding of the process of query generation requires understanding of the conceptual background of QTM. This background have been discussed in next chapter.

Chapter 3

Conceptual Background

This chapter provides the conceptual background necessary for the implementation of query and graph transformation manager QTM. First section describes the core concepts which are necessary to understand conceptual background of graph transformations. In the beginning after a brief introduction, the definitions of graph, meta model and instance model is provided. At the end of this section formal definition of graph transformation is provided.

This chapter includes a brief overview of PROGRES. PROGRES is given as an example of graph transformation system. As discussed earlier, currently, graph transformation can only be supported in DRAGOS using PROGRES. An example transformation is shown on the university example introduced in section 1.3.

Last section of this chapter includes a section about the concept of query transformation language (QTL). This section explains how this QTL can provide a universal solution and how existing application can integrate with QTM using QTL. It also includes details how QTM has been embedded in DRAGOS. At the end of this chapter a brief overview of the DRAGO's QTL called DRAGULA is included.

3.1 Graph Transformation

Graph transformations have already been introduced in chapter 1, so there is no need of repetition. The understanding of graph transformations require introduction to its basic concepts like graphs, meta model and instance model.

Graph

A graph consists of a set of vertices V and a set of edges E such that each edge e in E has a source and a target vertex s(e) and t(e) in V, respectively. Graphs provide the most basic mathematical model for entities and relations. graphs can represent concrete entities as vertices and relations between theses entities as edges.

Meta model and instance models can be best understood by comparing them to widely used UML notations. Unified modeling language (UML)[2] is the most popular modeling language which is used to model object oriented systems. The class diagram is the core of UML, which is good at describing static software architecture. UML is widely used to model object oriented in software engineering, it is assumed that reader is familiar with object oriented paradigm and basic UML notations such as class diagram, object diagram etc. When graphs modeled on object oriented principles, modeling occurs at two levels:

1. Type level, given by the class diagram. The model that describes the type level graphs is called the meta model.
2. Instance level, given by object diagram. The model that contains instance level graph is called an instance model.

3.1.1 Meta Model

Meta model is comparable to class diagram. Nodes in the meta model are called classes. A class may have attributes that define some kind of properties of the specific class. Inheritance may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra attributes. Associations define binary connections between classes (edge types between node types).

3.1.2 Instance Model

The instance model is a well formed instance of the meta model. Nodes and edges are called objects and links respectively. Objects and links are the instances of metamodel level classes and associations respectively. Attributes in metal model appear as slots in the instance model. Inheritance in the instance model imposes the instances of the subclass can be used in every situation where instances of the super class are required.

After having a look at the definitions of terms and concepts necessary to understand graph transformations. Its formal definition is provided in the section underneath.

Graph Transformation

Graph transformation [14] provides a pattern and rule based manipulation mechanism of graph models. Each rule application transforms a graph by replacing a part of it by another graph. A graph transformation rule $\mathcal{R}$ can be written as:

$$\mathcal{R} = (LHS, RHS, NAC)$$

Here, $\mathcal{R}$ contains a left-hand side graph LHS, a right-hand side graph RHS, and negative application condition graphs NAC. The LHS and the NAC graphs are together called the precondition PRE of the rule. and RHS are called post condition of the rule.

Rule application

The application of a rule to an instance model replaces a matching of the LHS in the model by an image of the RHS. Informally, this is performed by

- finding a match of the LHS in model

- checking the negative application conditions(which prohibits the presence of certain object and links)
- removing the part of the model that can be mapped to LHS but not to the RHS, yielding a context model
- replacing in context model, the parts found in earlier step with an image of RHS and at the end, a derived model is obtained.

Graph transformations in an application can also be supported by using relational databases as underlying implementation technology. This idea was presented by the authors of [41]. It involved following three salient points:

1. Creation of database schema based on the meta model
2. Implementation of pattern matching of rule using database queries
3. Handling of data manipulation

This thesis attempts to support graph transformations in DRAGOS based on its relational database support. An application using the developed QTM is going to create graph schema in DRAGOS. So, the first step is handled by DRAGOS itself. DRAGOS provides graph manipulation operations in its core services. These operations are used for data manipulation. So, DRAGOS API is used to do the third step. Coming to second step, implementation of pattern matching has been done by constructing SQL statement for the LHS and the NAC. The SQL statement is generated during parsing of QTL of DRAGOS which is called DRAGULA. The SQL generation is explained in detail in chapter 4. The explaination of QTL and DRAGULA is included in later sections of this chapter.

As already argued in chapter 1, currently DRAGOS supports graph transformations using PROGRES[33]. In next section a brief description of PROGRES is provided and later on it will be shown how DRAGOS uses PROGRES for graph transformation.

3.2 PROGRES

PROGRES (PROgramming with Graph Rewriting Systems) is a very high level language based on graph grammars, developed by Andy Schürr and Albert Zündorf of RWTH, Aachen in 1991. It is used to define, create, and manipulate graphs which are directed, typed, and attributed. It is available as free software. As The name suggests PROGRESS is for programming with graph rewriting systems.

PROGRES is a visual programming language having a graph-oriented data model: It has a graphical syntax for its most important language constructs. It was developed, according to its specification[34], having the following design goals in mind:

- Using graphical syntax where appropriate but at the same time not ignoring textual syntax when it can be more natural and concise.

- Distinguishing between data definition and data manipulation as database programming languages. Using and doing graph class declarations to type-check graph manipulations.
- Refrain users from the task to guarantee confluence of defined rewriting systems by keeping track of rewriting conflicts and backtracking out of dead-end derivations[34].
- Finally, not to always rely on the rule-oriented programming paradigm for all purposes but also supporting imperative programming of rule application strategies.

Use of PROGRES starts with definition of graph schemata. Next sections provides only broad overview of graph schemata and graph transformations using PROGRES, with the university example provided in section 3.2.2.

3.2.1 Definition of Graph Schemata

The idea of meta model has been explained before in section 3.1.1. Defining graph schema is identical to meta model. Nodes and edges are objects and relationships between these objects, respectively. In PROGRES, they are called node types and edge types respectively. Attributes are used to store the information that is local to a particular node. Edges cannot be attributed.

PROGRES offers the following syntactic constructs for defining the components of a particular class of graphs and their legal combinations. These are:

- **Node types**: which determine the static properties of nodes instances
- **intrinsic relationships**: which are called edge types, are explicitly concerned with the types of their sources and targets
- **derived relationships**: which model often needed paths of a given graph and which are defined by means of path expressions
- **intrinsic attributes**: which are defined for a particular set of node types and which are explicitly manipulated
- **derived attributes**: which are defined by means of directed equations and which may have different definitions for different node types

The graph schemata in PROGRESS for university example schema described in section 1.3 is shown in figure 3.1. It is read in following manner:

- Normal boxes represent node classes which are connected to their superclasses by means of dashed edges. In figure 3.1 professor, student, course, department and person are node classes.
- Boxes with round corners represent node types which are connected to their uniquely defined classes by means of doted edges.
- Solid edges between node classes represent edge type definitions; the edge type 'takes' is for instance a relationship between student node and course node.

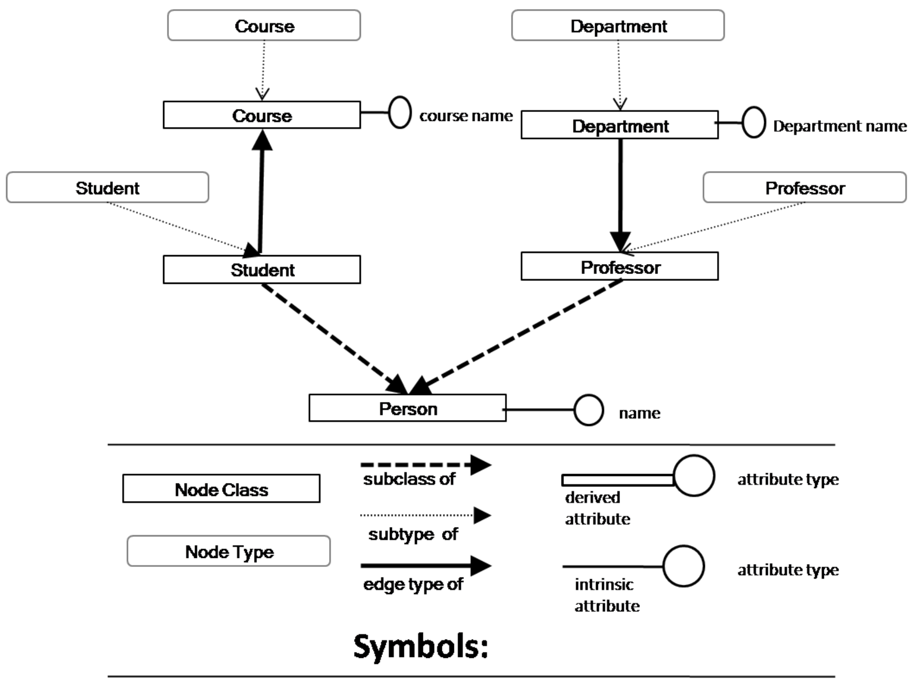


Figure 3.1: University Graph Schema in Progress

- Circles attached to node classes represent attributes with their names above or below the connection line segment and their type definition nearby the circle. A double line segment connects a derived attribute and a normal line segment connects an intrinsic attribute like 'Name' to its class.

3.2.2 Graph Transformations Using PROGRES

As already discussed in chapter 1, currently, in DRAGOS, graph transformations are only supported using code generated by PROGRES. This could be better explained by a real life example. As example University schema given in section 1.3 is taken. If a rule describing a professor employed by a department is to be transformed in to a professor starts teaching a course. The PROGRES syntax is given in figure 3.2. Here the so called left hand side (LHS) consisting of an edge 'employs' between two nodes 'department' and 'professor' is to be transformed in to so called RHS which constitutes of three nodes namely 'department', 'professor' and 'course' having two edges namely 'employs' between 'department'and professor and 'teaches between 'professor' and course respectively.

In this case, two nodes of type 'department' and 'professor' connected by edge of type 'employs' are queried. If this pattern is found in the graph storage, it is transformed corresponding to the rule's right-hand side (RHS, lower part). nodes assigned to '1 and '2 are preserved, as corresponding variables ('1 resp. '2) are present on the RHS. A new node of type 'course' is created and assigned to '3. As edges are neither identified nor attributed in the PROGRES graph model,

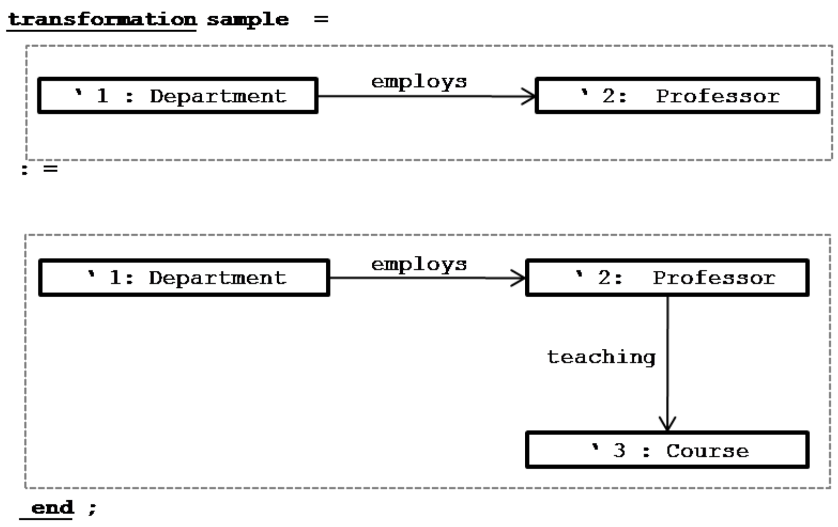


Figure 3.2: Example transformation rule in application-specific language

they do not need to be preserved explicitly: Removing all edges of the LHS and inserting edges corresponding to the RHS has the same effect as preserving edges if possible.

As a result of this thesis DRAGOS has been provided QTM which provides it an internal graph transformation mechanism. QTM parses a a pattern composed in DRAGOS's QTL. Pattern and QTL and discussed in next sections.

3.3 Pattern

A pattern, from the French patron, is a theme of reoccurring events or objects, referred to as elements of a set. These elements repeat in a predictable manner. A pattern in DRAGOS is written following DRAGOS's rule language called DRAGULA. as described in the figure 3.4, each Pattern consists of a set of PatternElements, which are sub-divided into Variables and Constraints. Constraints are connected to at least one Variable via Restricts edges, which can be distinguished using the role attribute. To support manipulation of graphs, the complete metamodel additionally provides Operators, which are not discussed in this document.

3.4 Query Transformation Language (QTL)

As elaborated in chapter 1, DRAGOS QTL provides tool support for existing graph languages. It acts as a base layer for graph languages and require developers to work at a higher level functionality provided rather than developing each time interpreters or code generation modules from scratch. QTL covers the entire DRAGOS graph model, e.g. querying nested graph structures and hyperedges are supported.

The Query and Transformation Language presented here, is able to represent rules on universal basis. By universal basis, it is meant that QTL is not limited to DRAGOS only. It means an application modeled in an application specific language has to convert its transformation rules in to the DRAGOS core language. Its done by importing the ASGs in to the graph database (e.g) by parsing a textual representation. The translation results in a graph structure representing a set of QTL rules, which are processed by the QTM's language implementation. Figure 3.3 shows how a graph transformation system modeled in an application-specific language interacts with DRAGOS QTM.

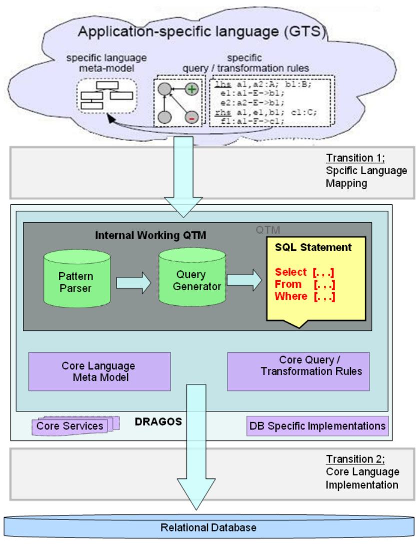

Application-specific language (GTS)
specific language meta-model
specific query / transformation rules
lhs a1,a2:A; b1:B;
e1:a1-E->b1;
e2:a2-E->b1;
rhs a1,e1,b1; c1:C;
f1:a1-F->c1;
Transition 1; Spcific Language Mapping
Internal Working QTM
QTM
Pattern Parser
Query Generator
SQL Statement
Select [. . .]
From [. . .]
Where [. . .]
Core Language Meta Model
Core Query / Transformation Rules
Core Services
DRAGOS
DB Specific Implementations
Transition 2; Core Language Implementation
Relational Database

Figure 3.3: Integration with the DRAGOS QTM

Figure 3.4 shows an excerpt of the meta-model of the core of QTL. A Pattern contains a set of PatternElements, i.e. Variables, Constraints and Operators. Variables represent the entities found during pattern matching. Variables and constraints are connected to each other via object of Restricts association. Constraints are used to put restrictions on assignments possible with Variables. Different constraints do different kind of restrictions. (E.g) Type constraint restrict assignment only to specific type, similarly containment restricts to a particular graph. The role attribute of the Restricts association describes the type of association between constraint and variable. For example, Incidence constraint determines from role attribute whether the attached variable is a source, target or connector.

An assignment of graph entities to variables of a Pattern not violating any of its Constraints is called a Match. Each Match is an aggregation of a set of Assignments, which relate a Variable to exactly one GraphEntity. It is required that each Variable with an attached Constraint has to be present in a Match, so partial matches are not allowed. Unconstrained Variables are not bound during pattern matching.

Operators define transformation of entities bound to a Match. Each Operator effects exactly one entity assigned to a Variable. For this purpose, operator require values of other variables as parameter. Required variables are distin-

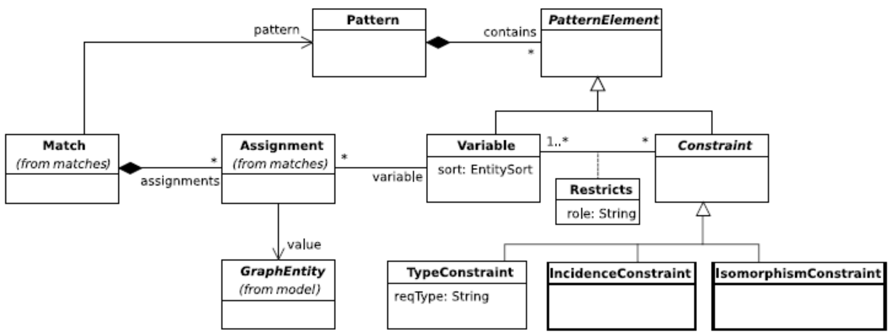


Figure 3.4: Meta-Model of the Query and Transformation Language

guished by a role name. Creation of an entity extend while deletion of an entity reduces the match by the effected variable. Operators are executed only when all required Variables are bound. This indirectly imposes an order on the operator's execution, as required variables are needed to be bound in advance.

The QTL for DRAGOS has been named DRAGULA and is presented in section 3.5. So, more details on constraints and variables are provided in that section. Next section shows how query and transformation mechanism has been embedded in DRAGOS. It also shows how existing applications can be integrated with DRAGOS to ensure the universal basis of QTL claimed earlier in this section.

3.5 DRAGOS Unified Language (DRAGULA)

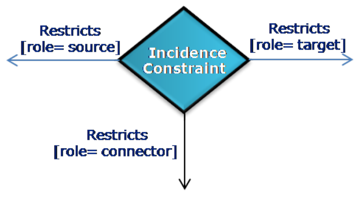
The Query and Transformation language developed for DRAGOS is called DRAGULA which is an abbreviation of DRAGOS Unified Language. DRAGULA is composed of constraints and variables.

3.5.1 Constraints

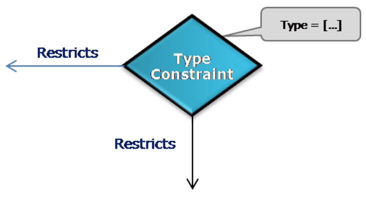
Constraints are used to put different type of restriction clauses in a pattern. There are six type of constraints available in DRAGOS. These are incidence, type, constant, identity, isomorphism and containment constraints respectively. These are discussed one by one in next sections.

Incidence Constraint

Incidence constraints is used to represent an edge. The source role variable points to the source of edge, the target variable points to the target of the edge and the concerned edge is pointed by the role Variable. Graphical representation of incidence constraint is provided in figure 3.5a.



(a) incidence constraint



(b) Type constraint

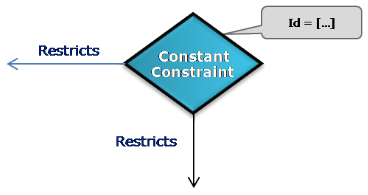
Figure 3.5: Incidence and Type Constraints

Type Constraint

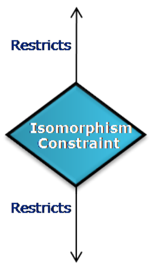
Type constraint is used to restrict entities to a particular type. This constraint can restrict both the edges and nodes. The type attribute indicates to the type to be restricted. Graphical representation of type constraint is provided in figure 3.5b.

Constant Constraint

Constant constraint is used to restrict entities to a specific entity. If search is to be restricted to a specific entity. Graphical representation of constant constraint is provided in figure 3.6a.



(a) Constant constraint



(b) Isomorphism constraint

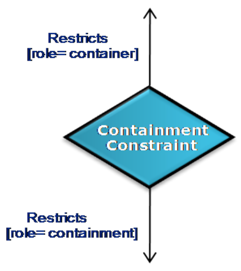
Figure 3.6: Constant and Isomorphism Constraints

Isomorphism Constraint

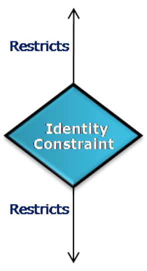
Isomorphism constraint is used to show that variables are related to each other. It shows the values of the variables are not equal. (e.g) let n1, n2 and n3 be connected via isomorphism constraint. it means n1 != n2, n2 != n3 and n1 != n3. Graphical representation of isomorphism constraint is provided in figure 3.6b.

Identity Constraint

Identity constraint is used to show that variables are related to each other. It shows the values of the variables are equal. Graphical representation of identity constraint is provided in figure 3.7b.



(a) Containment constraint



(b) Identity constraint

Figure 3.7: Identity and Containment Constraints

Containment Constraint

Containment constraint is used to represent nested datastructure. It has two role variables:

1. Container
2. Containment

Container points to the container which contains the variables pointed by containment variable. Graphical representation of containment constraint is provided in figure 3.7a.

3.5.2 Variables

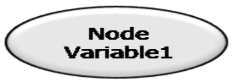
Nodes are represented by variables. Graph variables are used to represent graphs, edge variables are used to represent edges and nodes are represented by node variables. The pictorial representation of edge, graph and node variables is provided in figures 3.8a , 3.8b, 3.8c respectively.

Edge Variable1

(a) Edge Variable

Graph Variable1

(b) Graph Variable



(c) Node Variable

Figure 3.8: Variables

The conceptual background of QTM has been discussed in this chapter. Work cycle of QTM has been discussed in chapter 1, so, there is no need of repetition. After discussion of important elements of QTM such as pattern and QTL, next chapter explains how and on what principles QTM generates SQL statement.

Chapter 4

Query Generation

After the addition of QTM, DRAOGS supports graph transformation by generating SQL statement. This chapter describes details on SQL generation in QTM. It describes steps and phases involved in SQL generation. As already discussed a DRAGULA pattern is composed of node variables and constraints. In this chapter each constraint and node variable is described in terms of the SQL generation. An example pattern is taken on example schema 1.3 and then explained how SQL is generated in several phases. At the end, It gives a complete SQL statement for the example pattern.

Main jobs of a RDBMS is efficient storage and retrieval of data. Only the latter is relevant from QTM's perspective. Almost all of the modern day relational database management systems use different indexing and query optimization techniques to ensure efficient retrieval. When SQL query is generated, QTM assumes that RDMBS will ensure quick retrieval of records employing its optimization techniques. SQL is the standard manipulation language of relational databases. A typical SQL query is divided in to following three parts:

1. Select
2. From
3. Where

In QTM modifications are made at every phase in these three parts. At the end a complete SQL statement is prepared. For better understanding of this process, the example schema of section 1.3 is taken. The database design of DRAGOS has already been explained and sample insertions have been made in section 2.3.3. Several patterns can be executed on this schema, such as finding classmates of a particular student, checking whether a student is taken course of a particular professor or not, who are the professors from which a student is studying or courses taken by a student.

Suppose there is need to find class mates of a student susain as inserted in section 2.3.3. Class mates are the students, who have taken at least one same course as susain. it requires following two steps.

- In first step, find courses taken by a student.That means firstly look for edges of type 'takes' who have susain as source. The target field will give us courses taken by susain.

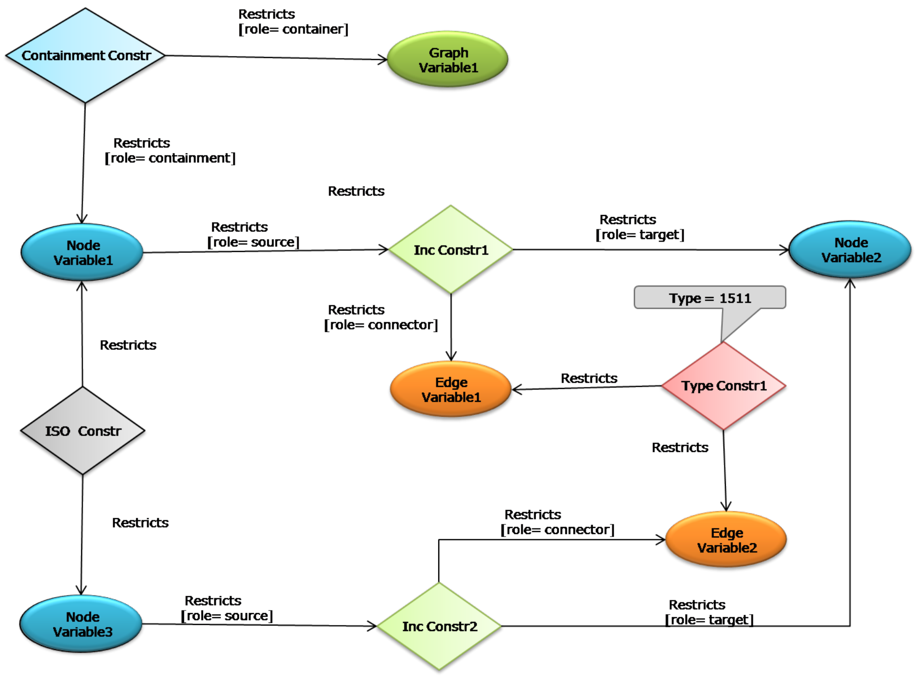


Figure 4.1: Class Mates

- In second step, look for the students who have taken these courses. Look for all 'takes' edges who have these courses in target field.

The Source field gives nodes which are class mates of a student susain. As already discussed in section 3.4, QTM needs patterns to be written in DRAGULA. The classmates pattern given in figure 4.1 finds class mates of susain.

The id attribute of constant constraint in figure contains 33405 as value. This is the value of attribute Id in table GraphEntity which represents susain in database. Incidence constraint1 has a node variable as its source and its target field contains all the courses taken by susain. The edge variable is restricted with edges of type 'takes'. Target field of Incidence constraint2 contains the courses taken by susain and source field give us the class mates. The edgevariable attached to the incidence constraint is also restricted by edges of type 'takes' via type constraint. as in source field node susain will be found too. Isomorphism constraint will ensure that node susain is not included among the classmates.

Next section provides a look at the modifications that are made on presence of different DRAGULA components. Class mates pattern is taken as example, modifications made are shown at each phase and at the end a complete SQL generated is given. As already described in description of DRAGULA in section 3.5 that it is composed of variables and constraints. Next section describes modifications made in SQL statement whenever variables are encountered.

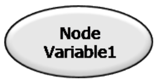


Figure 4.2: Node Variable

4.1 Variables

As already discussed in Section 3.5, DRAGULA is composed of various types of variables namely node, edge, graph, relation and relationend variables respectively. Edge variable and graph variables are treated differently than the rest. These two are described later in sections 4.1.2 and 4.1.1 respectively. Others variables node, relation and relationend are explained here. For each of the variables mentioned, following modifications are made in each of the three parts of a SQL statement:

1. Field 'id' is added in *SELECT* part of the SQL statement
2. Instance of table 'GraphEntity' is created in *FROM* part of the SQL statement
3. None in *WHERE* part

The working will be better understood by generating SQL for 'ClassMatePattern'. This pattern has three node variables. So, according to steps given above, three instances of table 'GraphEntity' should be created in *FROM* part of the SQL statement and 'id' field attribute associated with these instances should be added in *SELECT* part. After doing modifications, the SQL statement generated is as follows:

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1 ,
graphentity n2, graphentity n3
```

4.1.1 Graph Variable

On occurrence of a graph variable in a pattern following modifications are made in each of the three parts of a SQL statement:

1. Field 'id' is added in *SELECT* part of the SQL statement
2. Instance of table 'GraphEntity' is created in *FROM* part of the SQL statement
3. None in *WHERE* part

The 'ClassMatePattern' taken: has only one graph variable So, after following the steps given above, one instance of table 'GraphEntity' is created in *FROM* part of the SQL statement and 'id' field attribute associated with this instance is added in *SELECT* part. After doing modifications, the SQL statement generated is as follows:

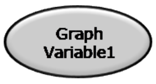


Figure 4.3: Graph Variable

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1 ,
graphentity n2, graphentity n3, graphentity g1
```

4.1.2 EdgeVariable

Edges are retrieved by edge variable. For each edge variable following modifications are made in SQL statement:

1. Field 'id' is added in *SELECT* part of the SQL statement
2. Instance of table 'GraphEntity' is created in *FROM* part of the SQL statement
3. None in *WHERE* part

Figure 4.4: Edge Variable

SQL generation for example 'ClassMatePattern' will help in understanding. This pattern has two edge variables. So, according to steps given above, two instances of table 'Edge' should be created in *FROM* part of the SQL statement and 'id' field attribute associated with these instances should be added in *SELECT* part. After doing modifications, the SQL statement generated is as follows:

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1,
graphentity n2, graphentity n3, graphentity g4,
edge e5, edge e6
```

4.2 Constraints

Constraints, as already described in chapter 4.2 are used to restrict values of the variables. In sections to follow, a brief overview of each constraint is provided. It is explained, how SQL statement is modified on occurrence of various constraints.

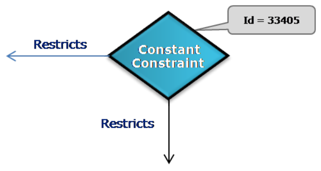


Figure 4.5: Constant Constraint

4.2.1 Constant Constraint

Constant constraint is used to restrict variables to a specific entity. Constant constraint is attached to the variables whose values are to be restricted to a single variable. Pictorial representation of a type constraint is given in figure 4.6. The Id attribute contains the database primary key of the entity who is to be restricted. For each constant constraint following modifications are made in a SQL statement:

1. None in *SELECT* part
2. None in *FROM* part
3. Id attribute of the processed 'ConstantConstraint' object is assigned to the 'id' fields of the attached objects of the variables. Database instances of the attached variables have already been on processing of variables as described in section 4.1. The 'id' field of the instances created are assigned the value contained in Id attribute of the constraint object.

Continuing with the 'ClassmatesPattern' example taken above, pattern contains one constant constraint. The id attribute contains 32935 which is value of the id field of graph entity that represents Susain in database. This value is the primary key of the student whose class mates are to be searched. After following the steps mentioned above, following SQL is generated:

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1,
graphentity n2, graphentity n3, graphentity g4,
edge e5, edge e6
WHERE
n1.id = 33405
```

4.2.2 Type Constraint

Type constraint is used to restrict entities to a particular type. This constraint can restrict all types of entities. Pictorial representation of a type constraint is given in figure 4.6. The value of 'Type' attribute contains the database primary key of the type to be restricted. For each type constraint following modifications are made in a SQL statement:

1. None in *SELECT* part

2. None in *FROM* part
3. type attribute of the processed 'TypeConstraint' object is assigned to field 'class' of table 'GraphEntity'.

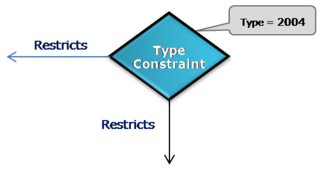


Figure 4.6: Type Constraint

Example pattern 'ClassmatesPattern' contains one type constraint restricting two edge variables. Instances of two edge variables in section 4.1.2 have already been created as e5 and e6 respectively. The 'class' field contained by these instances is assigned the value contained in type attribute of type constraint. The type attribute contains 2004 which is the value indicating the 'GraphEntityClass' id. This is the primary key of the graph entity class that represent student NodeClass in database. After following the steps mentioned above, following SQL is generated:

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1,
graphentity n2, graphentity n3, graphentity g4,
edge e5, edge e6
WHERE
n1.id = 33405 AND
e5.class = 2004 AND e6.class = 2004
```

4.2.3 Incidence Constraint

IncidenceConstraints demand connectivity of entities, using role names to distinguish between variables for the source, the target and the connector. Pictorial representation of a type constraint is given in figure 4.7. This distinction is necessary as DRAGOS allows edges to be connected to other edges, and so querying these structures needs to be supported. For each incidence constraint following modifications are made in SQL statement:

1. None in *SELECT* part
2. None in *FROM* part
3. The 'id' field of the instances of source and target variables of the processed 'IncidenceConstraint' object is assigned to 'source' and 'target' fields of table 'Edge' respectively. The selection of the instance of 'Edge' table is made on the basis of indication by the connector variable.

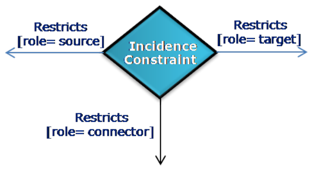


Figure 4.7: Incidence Constraint

'ClassmatesPattern' contains two incidence constraints attached to two edge variables. Instances of two edge variables in section 4.1.2 have already been created as e5 and e6 respectively. Process of creation of instances have already been explained in section 4.1. After following the steps mentioned above, following SQL is generated:

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1,
 graphentity n2, graphentity n3, graphentity g4,
 edge e5, edge e6
WHERE
n1.id = 33405 AND
e5.class = 2004 AND e6.class = 2004
AND e5.source = n1.id AND e5.target = n2.id AND
e6.target = n2.id AND e6.source = n3.id
```

4.2.4 Isomorphism Constraint

Isomorphism constraint is used to ensure that variables are related to each other. It shows the values of the variables are not equal. Pictorial representation of a isomorphism constraint is given in figure 4.7. For each isomorphism constraint following modifications are made in SQL statement:

1. None in *SELECT* part
2. None in *FROM* part
3. The objective of modification is to depict pairwise inequality of variables. So, in equality relation is created among the 'id' fields of the instances of the table 'GraphEntity' representing a variable. The mechanism can be better understood by an example. Let there be an isomorphism constraint attached to two node variables. The resultant SQL will be

```
Select [..] From [..],[..] where n1.id != n2.id
```

'ClassmatesPattern' contains a single isomorphism constraint attached to two node variables. Instances of these node variables in section 4.1 have already been created as n1 and n2 respectively. After following the steps mentioned above, following SQL is generated:

Restricts
Isomorphism Constraint
Restricts

Figure 4.8: Isomorphism Constraint

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1,
 graphentity n2, graphentity n3, graphentity g4,
 edge e5, edge e6
WHERE
 n1.id = 33405 AND
 e5.class = 2004 AND e6.class = 2004
AND e5.source = n1.id AND e5.target = n2.id AND
 e6.target = n2.id AND e6.source = n3.id AND
 n1.id != n3.id
```

4.2.5 Containment Constraint

Containment constraint is used to represent nested data structure. Pictorial representation of a containment constraint is given in figure 4.9. It has two role variables namely container and containment respectively. Container points to the container which contains the variables pointed by the containment variables. For each containment constraint following modifications are made in SQL statement:

1. None in *SELECT* part
2. None in *FROM* part
3. In where part, equality is ensured among 'parent' field of the instance of the table 'GraphEntity' of the variable attached to role variable containment and 'id' field of the instance of table 'GraphEntity' of the variable attached to the role variable containment. This could be better understood by an example. Let there be a containment constraint as shown in figure 4.9 and its attached to two node variables. The resultant SQL will be

   ```
   Select [..] From [..],[..]
   where n1.parent != n3.id and n2.parent = n3.id
   ```

The example 'ClassmatesPattern' followed so far contains a single containment constraint which is attached to a graph variable and a node variable.

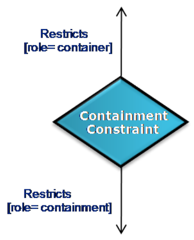


Figure 4.9: Containment Constraint

Instances of node and graph variable has already been created and explained in sections 4.1 and 4.1.1 as n1 and g4 respectively. After following the steps mentioned above, following SQL is generated:

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1,
 graphentity n2, graphentity n3, graphentity g4,
 edge e5, edge e6
WHERE
n1.id = 33405 AND
e5.class = 2004 AND e6.class = 2004
AND e5.source = n1.id AND e5.target = n2.id AND
e6.target = n2.id AND e6.source = n3.id AND
n1.id != n3.id AND
n1.parent = g4.id
```

Summary

After parsing the complete pattern and making all above mentioned modifications, a complete SQL statement is generated. DRAGOS uses template base code generation technique. This technique ensures that when underlying database is changed, there no requirement of change in source code. Instead only the template file is changed. The template file which generates SQL statement in QTM is included in Appendix(chapter 8) of this document. The SQL statement generated is then executed directly in RDBMS. The resultant graph is then returned by QTM. The complete SQL statement generated for example classmates pattern is given as follows:

```
SELECT n1.id, n2.id, n3.id FROM graphentity n1,
 graphentity n2, graphentity n3, graphentity g4,
 edge e5, edge e6
WHERE  n1.id = 33405 AND
e5.class = 2004 AND e6.class = 2004
AND e5.source = n1.id AND e5.target = n2.id AND
e6.target = n2.id AND e6.source = n3.id AND
n1.id != n3.id AND
n1.parent = g4.id
```

Chapter 5

RuntimeExperiments

This chapter includes the results of the runtime experiments conducted to evaluate the performance of QTM. In this chapter QTM's performance is compared with an implementation of same solution using in-memory version of DRAGOS and another version of DRAGOS which uses PROGRES for graph transformations. The schema presented in section 1.3 were taken as example. The results obtained after these runtime experiments are part of this chapter.

To evaluate performance of QTM three application were developed using different development options in DRAGOS. All these applications had similar schema given in section 1.3. All these applications were run on a computer with following specifications:

- **Operation System:** Linux
- **CPU:** Intel Core2 DUO, 2.6 GHZ
- **RAM:** 2 GB

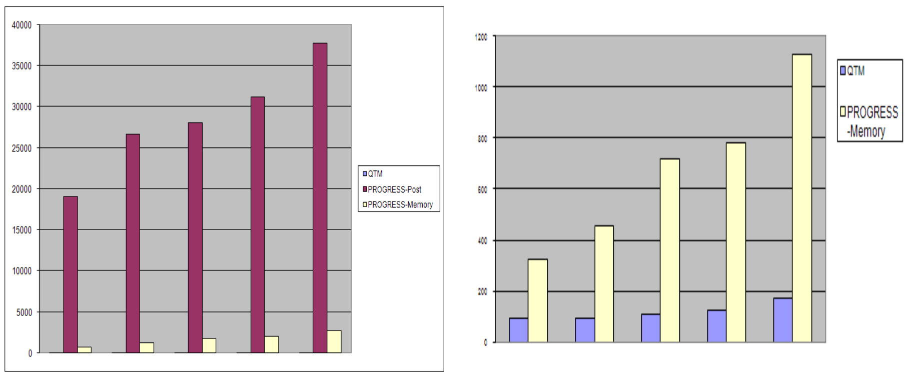
Similar patterns were executed on graphs of similar sizes in all these applications. Graphs of different graph sizes were inserted in database ranging from graph size of 2000 to 4000. At the end time measurements were taken to compare their performance with each other.

First application was implemented with DRAGOS containing QTM. PostgreSQL was selected as a storage back-end.

An application with university schema referred in section 1.3 was implemented in DRAGOS's in memory version which is the fastest among the DRAGOS's different back-end exchangeables. The in-memory version does not have a persistence storage. It keeps the data in volatile memory and when application is ended, the data is not saved to be retrieved later. This application used atomic operations provided by core services of DRAGOS to construct result. Due to its capability of storing data in main memory it provide the fast access among the different DRAGOS back-end exchangeables.

An application with similar functionality was implemented in DRAGOS's version before the addition of QTM. PROGRES was used in this version to support graph transformation operations.

Two patterns were executed. The executed patters were classmates and finding common students between the two departments specified. Class mates of a particular student are those students who have taken at least one course with the specified student. The second pattern looked for common students among the two departments specified. It means the students have taken courses in both the departments specified. The run time results obtained for pattern classmates and findcommonstudentsamongdepartments are shown in figures 5.1a and 5.1b respectively.



(a) Class Mates

(b) find common students among departments

Figure 5.1: Results of Runtime Experiments

The y-axis show the time taken in retrieval of graphs of different sizes. The scale used for time is milliseconds. The five bars from left show graphs of sizes 2000, 2500, 3000, 3500, 4000 respectively. The pattern shown in figure 5.1b display results only of in-memory version and QTM. The PROGRES version is excluded due to the fact that the difference of retrieval between PROGRESS and other two applications is tremendous. Its in factors of 100s and displaying that graphs results in ignorable time display of the other two applications. In figure 5.1a, retrieval time is not available for QTM because its value is so small as compared to PROGRES version that its not possible to display it in the chart.

The run time experiments reveal that QTM provides a performance edge by factors of 100s when its compared to traditional method of supporting graph transformations using PROGRES. The performance gain is even more in larger graph sizes. The fast in memory version of DRAGOS yields far better result than the application developed with using PROGRES. QTM, however, performs better than the in-memory version with a factor of 1:6. This factor reduces for small graph sizes but for larger graph sizes it increases.

Chapter 6

Related Work

In recent years lot of research has been carried out in investigating how graph transformations systems can be developed. They have been used in application areas such as model driven software development, service-oriented applications and pattern recognition techniques. A couple of them have been discussed in chapter 1, in this section a comparison will be drawn with work done related to this thesis. At the end of each section the work is compared with DRAGOS QTM.

6.1 Hybrid Query Language

In [3] Andries and Engels proposed Hybrid query language (HQL) including a method which translates hybrid queries into traditional textual queries by graph transformation. HQL queries are expressed by means of a mixture of graphical and textual elements. For graphical expression, they use extended version of the Entity Relationship model (named 'EER'-model). Based on this model, a formally defined SQL-like query language, named SQL/EER can be executed.

Brief description about the Extended Entity-Relationship Model (EER), SQL for the Extended Entity-Relationship Model and use of PROGRES is given in following sections.

6.1.1 The Extended Entity-Relationship Model (EER)

The Extended EntityRelationship (EER) model is based upon the entity relationship model[8] and according to its description[3], it is extended with the following concepts:

- Components: which are the object-valued attributes to model complex structured entity types
- Multivalued attributes and components to model association types
- Type construction in order to support specialization and generalization
- Several structural restrictions like the specification of keys, cardinality constraints, etc

6.1.2 SQL for the Extended Entity-Relationship Model

SQL/EER directly supports following concepts of the EER model:

1. Relationships, attributes of relationships, components and type constructions
2. Arithmetic
3. Aggregate functions
4. Nesting of the output
5. Subqueries as variable domains

Analogous to relational SQL which is discussed in chapter2, SQL/EER uses the select-from-where clause. SQL/EER supports sub-queries, inheritance and the use of relationship types as predicates.

Graphical alternatives are offered for the textual language constructs. These graphical alternatives do not replace their textual counterparts but are offered as alternatives, thus obtaining a hybrid query language . The language resulting from this extension is therefore called the hybrid query language HQL/EER . Briefly, a query in HQL/EER consists of an attributed labeled graph, a piece of text (SQL/EER). In case of only textual part, the graph generally consists of declarations, conditions, as well as selections.

6.1.3 Use of PROGRES for expressing graph model

PROGRES which is already discussed in 3.2 is used by HQL to express its graph model. Details of term graph model can be seen in section 2.2.2. According to specification of HQL[3], the entire specification of the graphical part of HQL/EER is made using the PROGRES. The specification is entered using the PROGRES system syntax directed editor which allows the specification to be analyzed by the system's incrementally working type-checker and executed by the system's integrated interpreter . The interpretation (the execution of a sequence of productions) is generated using the PROGRES system. The graphical part of HQL/EER queries are formalized by means of a PROGRES specification in following two step process:

1. The syntactic structure of the graphical part of HQL/EER queries is captured in a PROGRES specification
2. The semantics are defined from the specification. The specification is extended with additional node attributes and attribute derivation rules. These rules translate the graphical part of the HQL/EER query into a SQL/EER query which is combined with the textual part of the hybrid query into a full SQL/EER-query.

QTM generates SQL statement for every pattern composed in DRAGULA pattern and this SQL statement is used for pattern matching and performing graph transformations. HQL on the other hand uses graph transformation to convert hybrid queries in to textual queries. PROGRES tool is used for graph model storage and operations. The conversion of hybrid queries in to textual queries is also performed using PROGRES. On the other hand QTM is internal to DRAGOS and it is not dependent on any other application specific transformation language. One more worth mentioning difference between HQL and DRAGULA is that HQL is conceptually very close to SQl which eases the translation while DRAGULA is not directly related to SQL. DRAGULA, however, adopts the concepts of predicates.

6.2 Graph Transformation engine in RDBMS

Implementation of a graph transformation engine using standard relational database management systems (RDBMSs) was first suggested by Gergely Varró and his colleagues in [42]. They suggested various steps to develop a graph transformation engine in relational database. The steps are:

1. **Mapping metamodel to database tables:** which is to generate the schema of the database from the metamodel

2. **Creating database representation of instance models:** which is storing the instance models representing the system in the database tables.

3. **Creating database Views for LHS and NAC:** which means to matching patterns of a graph transformation rule by using views which contain all matchings of the rule. They suggested a separate view for each LHS and NAC graph. They proposed that when the view for the precondition graph is calculated, views of all its positive and negative application conditions are available. If the precondition has no negative application conditions then the view defined for the LHS contains the database representation of all matchings of the precondition graph.

4. **Model manipulation in relational databases:** it is done by operations in the graph manipulation phase are implemented by issuing several data manipulation commands (INSERT,DELETE, and UPDATE) in a single transaction block. The transaction block is needed to ensure that a graph transformation step is atomic, i.e., either all commands or none of them are executed to ensure a consistent model after rule application.

First difference between the above mentioned approach to DRAGOS is that they suggest a conversion of meta model in to relational database tables for every graph application while DRAGOS graph model constitutes a common meta model for all applications. Usage of a new application does not result in generation of new database tables in DRAGOS. QTM generates SQL code from the DRAGULA pattern while in above mentioned approach SQL is generated on the basis of application rules stored in database. The concept of query transformation language is not present in Varró's work while QTM generates SQL after parsing query transformation language of DRAGOS called DRAGULA. Varró constructs database views for application rules while QTM generates a single SQL statement for every complex pattern. Graph replacement in QTM is done by using operations provided by DRAGOS core API while Varró uses SQL statements to replace graphs.

6.3 Unified Data Model (UDM)

The UDM (Universal Data Model) framework defines a development process and a set of supporting tools that are used to generate programmatic interfaces from UML class diagrams. The working of UDM can be better explained by figure 6.1.

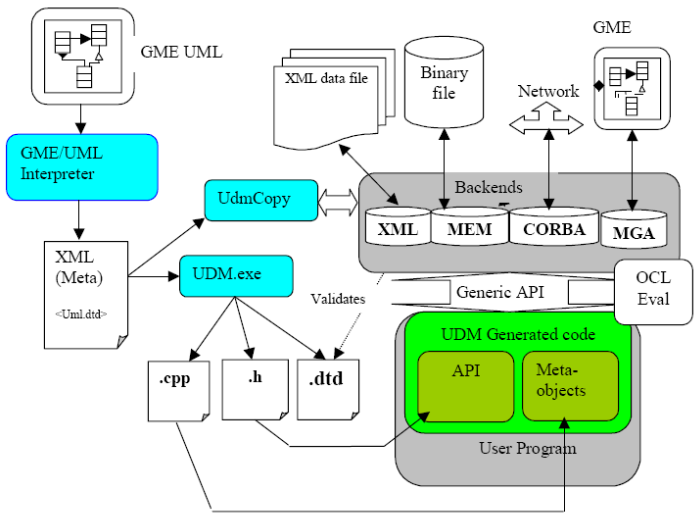


Figure 6.1: UDM tools and architecture *reproduced from*[24]

As described in [24], the first step while using the UDM framework is to create a UML class diagram in the its GME/UML modeling environment. The second step is to interpret the class diagram by using a GME [11] model interpreter. This interpreter module traverses the class diagram and generates a corresponding XML description of the UML classdiagram. This xml description is referred as as Metamodel. The third step is to generate the API code for the Metamodel. it is done by the 'Udm.exe' tool. This tool reads the input XML and generates corresponding .h, .cpp, and .dtd files.

UDM relies on a limited set of base classes while DRAGOS uses a complex graph model for data representation. UDM generates APIs from provided 'Metamodel', this functionality is not provided by DRAGOS. Like DRAGOS, UDM environment provides persistent storage using databases through the Generic Modeling Environment (GME) [11]. UDM does not incorporate a model processing engine, but can be used by the GReAT transformation engine[1].

Chapter 7

Conclusion and Future Work

As per requirement, this thesis adds graph transformation support in DRAGOS graph database. The developed graph transformation is based on DRAGOS's relational database support. It implements a graph transformation by a Query and Transformation Mechanism (QTM). The idea of developing a graph transformation system based on the relational databases is not new. QTM implements a transformation mechanism which uses a query transformation language (QTL) which provides an extensible and adaptable graph language with a standardized mechanism to access access RDBMS.

This transformation language acts as a building block for various kinds of graph languages which allows developers to work at a higher level functionality. Integrations of external applications with QTM is performed by converting their language-specific rules in to DRAGOS's QTL called DRAGULA. Most critical operation of a typical graph transformation system, pattern matching is performed by generating a single SQL statement after parsing a pattern in DRAGULA. The problem with SQL is that its syntax varies from database to database.

QTM's deficiency of generating RDBMS specific SQL is overcame by template based code generation. Open source tool for this purpose called Velocity has been used. The template based code generation ensures separation between program and code. It means that changing database design or database does not require modification in DRAGOS code rather only it requires change in template file.

QTM provides a significant performance gain. The runtime experiments conducted during course of this thesis have cemented this performance gain. The experiments were conducted to compare performance of QTM with a similar functionality versions developed with GRAS, older version of DRAGOS with PostgreSQL support and with fast in-memory version. The experiments conducted showed that QTM has significantly outperformed its rivals. This thesis adds a fast and robust transformation engine in DRAGOS.

Future Work

As a future extension of this thesis work, number of extensions are proposed. Currently, QTM does not support attribute based evaluation and data transformation. Operators in DRAGULA are currently not handled by QTM. These are the extensions that could be made in current system. Moreover, to handle interaction between graph transformation system, the control structures for handling iterations and conditional branching should be introduced. All these features were not mandatory for the current QTM extension of DRAGOS and they remain as future extension.

Chapter 8

Appendix

```
/** One variable each for three parts of SQL query */

#set ($selectQuery = "")
#set ($fromQuery = "")
#set ($whereQuery = "")

/** Code generation of Node variables **/
/** The QTMmap is the object of java class java.util.map*/
/** The helper.int method returns an integer which is
    incremented by one on each call to helper.int
*/
/** Node list object contain all the Node variables */

#foreach( $item in $nodeList )
      #set ($selectQuery = "$selectQuery  ,n$helper.int .id" )
      #set ($fromQuery = "$fromQuery ,GraphEntity as n$helper.int")
      $qtmMap.addNode($item,"n$helper.int ")
  $helper.next
#end

/** Code generation of edge variables */
/** edgelist contains list of all edge variables. */

#foreach( $item in $edgeList )
      #set ($selectQuery = "$selectQuery  ,e$helper.int .id")
      #set ($fromQuery = "$fromQuery ,GraphEntity as e$helper.int")
      $qtmMap.addNode($item,$"e$helper.int")
   $helper.next
#end

/** Code generation of Incidence Constraint */
/** incidenceList contains list of all incidence constraints. */
/** item.connecter gives attached variable with role connector */
/** item.source gives attached variable with role source */
/** item.target gives attached variable with role target */
/** .alias gives database alias attached the variable.
*   note: this alias was saved in mapped earlier.
*/

#foreach( $item in $incidenceList )
  #if($qtmMap.isThereVariable($item.connector))
     #set( $var = "$qtmMap.getVariable($item.connector).alias")
                 #set ($whereQuery="$whereQuery and $var.source=
                   $qtmMap.getVariable($item.source).alias.id and
        $var.target=
                     $qtmMap.getVariable($item.target).alias.id")
  #end
    $helper.next
#end

/** Code generation for Type Constraints */

#foreach( $item in $typeList )
  #foreach( $item1 in $item.constrainedVariables )
    #set( $var = "$qtmMap.getVariable($item1).alias")
    #set ($whereQuery = "$whereQuery and $var .class
     = $item.getTypeId()" )
        #end
#end

/** Code generation for Constant Constraints */

#foreach( $item in $constList )
  #foreach( $item1 in $item.constrainedVariables )
    #set ($whereQuery = "$whereQuery and
     $qtmMap.getVariable($item1).alias.id= $item.id")
  #end
#end

#* Isomorphism Constraint *#

#foreach( $item in $isoList )
  #foreach( $item1 in $item.allVariables )
    #if($qtmMap.isThereVariable($item1.variable1))
    #set ($whereQuery = "$whereQuery and
    $qtmMap.getVariable($item1.variable1).alias.id!=
     $qtmMap.getVariable($item1.variable2).alias.id")
     #end
  #end
#end

#* Identity Constraint *#

#foreach( $item in $identityList )
```

```
  #foreach( $item1 in $item.allVariables )
    #if($qtmMap.isThereVariable($item1.variable1))
     #set ($whereQuery = "$whereQuery and
     $qtmMap.getVariable($item1.variable1).alias.id=
     $qtmMap.getVariable($item1.variable2).alias.id")
    #end
  #end
#end

#* Containtment Constraint *#

#foreach( $item in $contList )
  #foreach( $item1 in $item.constained)
   #set ($whereQuery = "$whereQuery and
    $qtmMap.getVariable($item.cotainer).alias.parent =
     $qtmMap.getVariable($item1).alias .id" )
  #end
#end
```

Bibliography

[1] A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Journal on Software and System Modeling*, 5(3):261–288, September 2006.

[2] Sinan S. Alhir. *UML in a Nutshell (In a Nutshell)*. O'Reilly, September 1998.

[3] Andries and G. Engels. A hybrid query language for an extended entity-relationship model. *Journal of Visual Languages and Computing*, 7(3):321–352, September 1996.

[4] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Sci. Comput. Program.*, 34(1):1–54, 1999.

[5] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1–39, 2008.

[6] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 402–429, London, UK, 2002. Springer-Verlag.

[7] Boris Böhlen and Ulrike Ranger. Concepts for specifying complex graph transformation systems. In Ehrig et al. [14], pages 96–111.

[8] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[9] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 404–416, New York, NY, USA, 1990. ACM.

[10] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. Viatra: Visual automated transformations for formal verification and validation of uml models, 2002.

[11] James Davis. Gme: the generic modeling environment. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–83, New York, NY, USA, 2003. ACM.

[12] Juan de Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 174–188, London, UK, 2002. Springer-Verlag.

[13] Germany Department of Computer Science 3 RWTH Aachen University, Aachen. Official documentation of dragos. `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp ?topic=/com.ibm.db2.udb.admin.doc/doc/c0004100.htm`.

[14] Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors. *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*. Springer, 2004.

[15] Frank Ch. Eigler. Translating graphlog to sql. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 14. IBM Press, 1994.

[16] International Organization for Standardization. Structured query language (sql). `http://www.iso.org/`, 2008.

[17] Reiko Heckel. Graph transformation in a nutshell. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[18] Richard C. Holt, Andreas Winter, and Andy Schürr. Gxl: Toward a standard exchange format. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 162, Washington, DC, USA, 2000. IEEE Computer Society.

[19] American National Standards Institute. Structured query language (sql). `http://www.ansi.org/`, 2008.

[20] N. Kiesel, A. Schürr, and B. Westfecthtel. Gras: a graph-oriented software engineering database system. In *Building tightly integrated software development environments: the IPSEN approach*, pages 397–425, New York, NY, USA, 1996. Springer-Verlag New York, Inc.

[21] Anja Klein, Rainer Gemulla, Philipp Rösch, and Wolfgang Lehner. Derby/s: a dbms for sample-based query answering. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 757–759, New York, NY, USA, 2006. ACM.

[22] Manu Konchady. An introduction to jdbc. *Linux J.*, page 2, 2005.

[23] International Business Machines(IBM). Structured query language (sql) (html). `http://www-i3.informatik.rwth-aachen.de/research/dragos/`, 2008.

[24] Endre Magyari, Arpad Bakay, Andras Lang, Tamas Paka, Attila Vizhanyo, Aditya Agrawal, and Gabor Karsai. Udm: An infrastructure for implementing domain-specific modeling languages, October 2003.

[25] M. Minas and G. Viehstaedt. Diagen: a generator for diagram editors providing direct manipulation and execution of diagrams. In *VL '95: Proceedings of the 11th International IEEE Symposium on Visual Languages*, page 203, Washington, DC, USA, 1995. IEEE Computer Society.

[26] Bruce Momjian. *PostgreSQL: introduction and concepts.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[27] Manfred Münch. Programmed graph rewriting system progres. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 441–448, London, UK, 2000. Springer-Verlag.

[28] Giuseppe Naccarato. Template-based code generation with apache velocity. `http://www.onjava.com/pub/a/onjava/2004/05/05/cg-vel1.html`, 2004.

[29] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The fujaba environment. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 742–745, New York, NY, USA, 2000. ACM.

[30] Matthew Norman. *Database Design Manual: using MySQL for Windows (Springer Professional Computing).* Springer, September 2003.

[31] Arend Rensink. GROOVE: A graph transformation tool set for the simulation and analysis of graph grammars, 2003.

[32] Grzegorz Rozenberg, editor. *Handbook of graph and computing by graph transformation: volume I. foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[33] A. Schfürr. Programmed graph replacement systems. In *Handbook of graph grammars and computing by graph transformation: volume I. foundations*, pages 479–546, River Edge, NJ, USA, 1997. World Scientific Publishing Co., Inc.

[34] Andy Schürr. Progres for beginners.

[35] Bengt Sigurd. Implementing the generalized word order grammars of chomsky and diderichsen. In *Proceedings of the 13th conference on Computational linguistics*, pages 336–340, Morristown, NJ, USA, 1990. Association for Computational Linguistics.

[36] Mistry Harjinder Singh and Srinath Srinivasa. Grace: A graph database system. *International Conference on Management of Data COMAD*, 2005.

[37] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *AGTIVE*, pages 481–488, 1999.

[38] Toby J. Teorey. *Database Modeling & Design.* Morgan Kaufmann, January 1999.

[39] J. D. Ullman. *Principles of Database Systems, 2nd Edition.* Computer Science Press, Rockville, MD, 1982.

[40] Gergely Varró. Implementing an ejb3-specific graph transformation plugin by using database independent queries. *Electr. Notes Theor. Comput. Sci.*, 211:121–132, 2008.

[41] Gergely Varró, Katalin Friedl, and Dániel Varró. Graph transformation in relational databases. *Journal of Software and Systems Modelling*, 5(3):313–341, September 2006.

[42] Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a graph transformation engine in relational databases. *Software and System Modeling*, 5(3):313–341, 2006.

[43] Erhard Weinell. Adaptable Support for Queries and Transformations for the DRAGOS Graph-Database. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Proc. of the* 3^{rd} *Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*, volume 5088 of *Lect. Notes in Comp. Sci.*, pages 369–411. Springer, 2008.

[44] Erhard Weinell. Extending graph query languages by reduction. In Claudia Ermel, Reiko Heckel, and Juan de Lara, editors, *Proc. of the* 7^{th} *Intl. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'08)*, volume 10 of *Elec. Communications of the EASST*, 2008.