GRAPHLOG: ITS REPRESENTATION IN XML AND TRANSLATION TO CORAL

Liqian Zou

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

April 2003

© Liqian Zou , 2003

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-78004-X

Canada

Abstract

Graphlog: Its Representation in XML and Translation to Coral

Liqian Zou

Diagrammatic Query allows users to compose a query through diagrams in a Graphical User Interface(GUI) environment. Diagrammatic Query has three basic requirements that must be fulfilled:

- Both the query and query results should be presented graphically.
- The users should be able to query database intuitively.
- Queries should not only refer to information that resides at databases, but also to views based on defined queries, defined previously, which are saved in the system.

We utilize the logic of Graphlog and CORAL so that users are able to pose queries by drawing nodes, edges and blobs, and probably by specifying which existing views are to be included.

This thesis proposes the definition of Transferable Graph Language and the analysis, design, and implementation of the translation system. The transformation and storage format used by our project is specified in Transferable Graph Language(TGL), which uses XML DTD to regulate the program structures. The translation system will be driven by TGL programs transferred from the GUI system. These TGL programs collect node, edge, and blob status and information about how they are related. Also they will inform which existing views will be involved. Then the translation system will analyze these data and translate them to CORAL programs. Next, the translation system will organize calls to CORAL system and collect query results received from the CORAL system. At the end, the translation system will transform the results as TGL result programs and return them to the GUI system for displaying to users.

Acknowledgements

Above all, I wish to express my deepest appreciation to my thesis supervisor, Dr. Gregory Butler. His inspiring and patient guidance lead me to accomplish my thesis. I will never forget the principle that he taught me: Keep things simple. I can not remember how many times this principle sends me on my way to conquer difficulties and derive elegant solutions.

In addition, my gratitude would go to Xuede Chen and Helen Lin, who worked together with me on the same project. They gave me valuable suggestions.

Continuely, I am alway grateful to my mother, it is her love that is accompanying me through the master study.

At the end, I greatly appreciate the love, support, and encouragement of my husband, Mang Yu.

Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Deductive Database Systems 1
1.2 Graph Query Languages and Support Systems 2
1.3 The Role of the Thesis in the Whole Diagrammatic Query System . . 2
1.4 Contribution of the Thesis 3
1.5 Organization of the Thesis 4

2 Background 5

2.1 Object-Oriented Incremental Process Model 5
2.2 Graphlog and Hygraph 6
2.2.1 Fundamental Elements 6
2.2.2 Aggregate Functions 9
2.2.3 Graphlog, SQL and Object-Oriented Query Language 10
2.3 CORAL system 11
2.3.1 Architecture of CORAL Deductive System 11
2.3.2 CORAL Declarative Language 12
2.3.3 The Query Optimization 14
2.4 Extensible Markup Language(XML) 14
2.4.1 General Definitions 15
2.4.2 The relation of XML with SGML and HTML 15
2.4.3 XML Constructs 16
2.4.4 XML Processors 17

2.5 Java . . . 18
2.5.1 The Architecture of Java . . . 18
2.5.2 Collections Framework [20] . . . 19
2.5.3 The Input/Output Framework . . . 22

3 Transferable Graph Language **25**
3.1 Lexical Elements . . . 26
3.1.1 Characters used . . . 26
3.1.2 Identifiers . . . 26
3.1.3 Primitive Types and Literals . . . 27
3.1.4 Operators . . . 27
3.1.5 Expressions . . . 29
3.2 Query Program Structure . . . 31
3.2.1 Program Elements . . . 31
3.2.2 Content Models . . . 33
3.3 Result Structure . . . 33
3.3.1 Program Elements . . . 35
3.3.2 Content Models . . . 37
3.4 Comments . . . 37

4 Translation System **39**
4.1 Context of The Translation System . . . 39
4.2 Requirements and Incremental Plan . . . 41
4.2.1 Requirements Definition . . . 41
4.2.2 Incremental Plan . . . 48
4.3 Increment One Process . . . 52
4.3.1 Increment One — Analysis . . . 52
4.3.2 Increment One — Design . . . 57
4.3.3 Increment One — Implementation . . . 73
4.3.4 Increment One — Test . . . 77
4.4 Increment Two Process . . . 83
4.4.1 Increment Two — Analysis . . . 83
4.4.2 Increment Two — Design . . . 87
4.4.3 Increment Two — Implementation . . . 96

4.4.4 Increment Two — Test 100
4.5 Increment Three Process 105
4.5.1 Increment Three — Analysis 105
4.5.2 Increment Three — Design 108
4.5.3 Increment Three — Implementation 117
4.5.4 Increment Three — Test 119

5 Conclusion **127**

Biblography **130**

Appendixes **133**

A University Data Model System Schemes and Example Facts **133**

B Sample Query Diagram Templates **139**

List of Tables

1 A Comparison of XML, SGML and HTML 16
2 The Arithmetic Operators Applicable to TGL 28
3 The conjunction Operators Applicable to TGL 29
4 The Content Model of TGL Query Program 34
5 The Content Model of TGL Query Program(continue) 35
6 TGL Result Content Model 37
7 Tokenizer States and Associated Characters 66
8 Newly Declared Exception Classes Serving Increment One 73
9 The System Scheme File shared with lower layer system 80
10 The CORAL Program That Define Relation dept_people 102
11 The Coral Program That Query Using a TGL logical Expression . . . 121
12 The Result TGL Program For a Query Using a TGL logical Expression 122
13 The Coral Program That Query Involving Negation 124
14 The TGL Program for Results of the Query Involving Negation . . . 125
15 The Coral Program That Query Involving Aggregation 126
16 The TGL Program for Results of the Query Involving Aggregation . . 126
17 The Underlying System Scheme for the Univesity Data Model 134

List of Figures

1 The Incremental Process Model 7
2 An Example of DefineGraphlog 9
3 Examples of Defining and Using Blobs 10
4 The Architecture of CORAL System 11
5 Core Collections Interface Hierarchy of Java 20
6 Character Stream Class Hierarchy of Java 23
7 Byte Stream Class Hierarchy of Java 24
8 The Two Methods to Depict Attributes 29
9 TGL Logical Expression Syntax 30
10 The TGL Program Structure Regulation 32
11 The TGL Result Structure Regulation 36
12 Context Diagram of The Translation System 40
13 Use Case Diagram Inside The Translation System 42
14 Activity Diagram of The Translation System 49
15 The Analysis Class Diagram of Increment one 58
16 The Conceptual Structure at Stage of Increment One 58
17 Organization of Nodes, Edges, and Blobs Before Applying Flyweight Pattern 62
18 Organization of Nodes, Edges, and Blobs After Applying Flyweight Pattern 63
19 Nested Class 63
20 Inner Class XRuntime in Increment One 64
21 Tokenizer States Class Hierachy 66
22 Potential Paths of recognizing a Symbol 67
23 XToken Class Diagram 68
24 Tokenizing TGL Programs Class Diagram 69

25 Class Hierarchy for Parsing TGL Program 71
26 An Example Coral Program 72
27 The design class diagram of Increment one 74
28 The Implementation of Class LocalTranslator Being Singleton 75
29 Test Object for Fetch System Schemes 81
30 Test case result of Fetch System Schemes 82
31 The Analysis Class Diagram of Increment Two 87
32 The conceptual structure at stage of Increment Two 88
33 Updated Inner Class XRuntime 89
34 Accessing CORAL System 90
35 An example of CORAL Backend Output 92
36 Collaborating to Access the CORAL System 93
37 The Sequence Diagram for Parsing Result 94
38 The design class diagram of Increment Two 95
39 Implementation of Recursively Recording Related User-defined Relations 97
40 Implementation of Combining Collection without Repetition 97
41 Implementation of Calculating Number of Generation 98
42 Implementation of Listing All Based User Schemes In order 99
43 Drawing of Query "Return Staff Members and Students in the Computing Science Department" 101
44 TGL Program - Return Staff members and students in the Computing Science Department 103
45 CORAL Execution Results - Return Staff members and students in the Computing Science Department 104
46 TGL Program Corresponds to the CORAL Execution Results 106
47 Final Conceptual Structure of the Translation System 109
48 Translation TGL logical Expression Class Diagram 110
49 Class Diagram of PredicateParser 113
50 The Context-free Grammar for TGL logical Expressions 113
51 ASTNode Constructions Examples 114
52 Design of Class ASTNode 115
53 Design of Class CodeGenerator 115
54 Design of classes XNode, XEdge and XBlob 116

55 Implementation of Detail of An Edge 118
56 Graph&TGL Program for an Query Example Using Positive TGL logical Expressions 120
57 Graph for an Query Example Involving Negation 123
58 Graph for an Query Example Involving Aggregation 125
59 Diagrams of Example Query Composition(1 – 6) 140
60 Diagrams of Example Query Composition(7 – 11) 141
61 Diagrams of Example Query Composition(13 – 16) 142
62 Diagrams of Example Query Composition(17 – 22) 143
63 Diagrams of Example Query Composition(23 – 26) 144
64 Diagrams of Example Query Composition(27 – 30) 145

Chapter 1

Introduction

Along with the development of database technologies, great efforts has contributed to the field of inventing query systems, making it easier for end users to compose expressive, efficient, and powerful queries.

In fact, various query systems have been designed and implemented to assist corresponding database data models. This thesis deals with diagrammatic queries that are based on Graphlog [3] and CORAL [6], an excellent deductive database.

1.1 Deductive Database Systems

Deductive database systems have been under extensive research since 1980s. Until the late 90s, powerful deductive database systems were created. The CORAL deductive system [6] is such an example.

Deductive database systems are extended relational database systems. As relational database technology has matured and related optimizers have been improved over the last decades, relational systems have become more widely used. However, relational data model has painful limitations for scientific applications such as bioinformatics [1]. Above all, it can not process recursive queries, which are normal requirements of most scientific applications. In effect, deductive database is not only an extension of relational database, but an extension of logic programming languages like Prolog [2]. So deductive databases inherit both the declarative features of relational database query languages and rule-based query ability from logic programming

languages. Thereby, a deductive database has potentially the high performance of relational database and flexibility on query description provided by logic programming systems. The CORAL system developed at University of Wisconsin is one of the most practical implementations of deductive database.

Although declarative query languages of deductive databases already have significant power for end users to express the queries, they are still less expressive compared with graph query languages.

1.2 Graph Query Languages and Support Systems

"A picture is worth a thousand words". Graph query languages use diagrams as standard user interface. Compared with graph queries, deductive queries are much less expressive and less easy to use. The Graphlog query language and its querying environment, Hy+ [5], were designed and implemented in University of Toronto.

The Hy+ system utilizes the CORAL deductive system as its underlying support to process queries [4].

There are several prototype applications that make use of Graphlog and Hy+ [4]. Nonetheless, the querying environment Hy+ system is implemented using Smalltalk, which at least limits its integration possibility with other well-used applications.

1.3 The Role of the Thesis in the Whole Diagrammatic Query System

Our project of designing and implementing a diagrammatic query system for use with biological projects are basically composed of three parts: the Graphical User Interface(GUI) system for biological scientists to compose query intuitively, the back-end database system to support the CORAL deductive system for executing queries, and the translation system residing between the GUI system and the CORAL deductive database system.

This thesis is responsible for the design and implementation of the translation system. Also it defines a textual language to record and transfer queries and query results between the GUI system and the CORAL system.

1.4 Contribution of the Thesis

This thesis aims to provide support for diagrammatic queries oriented to scientific applications such as Bioinformatics. First, it defines the Transferable Graph Language. Second, it designs a translation system that bridges between graph query interface systems like Hy+ and the CORAL deductive system. The implementation is realized using the Java programming language.

Through these efforts, this thesis contributes the following aspects:

- extends Graphlog by providing a family of built-in predicates.
- provides suggestions about how to implement the GUI system where the graph queries are drawn. Through the regulation of its Transferable Graph Language and the examples presented, the thesis proposes the drawing elements and utilities.
- invents a way to write transferable textual records of the graphs. The Transferable Graph Language makes use of XML Data Definition Types as programming templates. As a consequence, the programs generated can be utilized in many applications across different platforms.
- presents an incremental development process that has advantages over waterfall and iterative process models.
- conforms to object-oriented development, thereby promoting reusability.
- creates a platform-independent translation system. Because the translation system is implemented in Java, it is able to cooperate with applications written in different languages and reside in different platforms.
- provides query templates. Through the examples in test cases and appendix, the thesis provides a large number of templates representative of typical queries.
- processes textual queries. Although the purpose of designing the Transferable Graph Language is to record graph queries, queries can be constructed directly using the TGL templates and fetch results. In effect, the translation system has been integrated with the CORAL system. I is a useful application by itself.

1.5 Organization of the Thesis

This thesis comprises five Chapters and two appendixes.

Chapter 1 introduces the origin of the thesis and what the thesis contributes.

Chapter 2 provides background knowledge for the thesis. Object-oriented development techniques, a graph query language called Graphlog and its supporting system Hy+, the CORAL deductive system, Extensible Markup Language, and Java are described in detail. The description mainly focuses on the knowledge that directly relates to the thesis.

Chapter 3 presents the Transferable Graph Language that is defined in this thesis. This language specifies graph elements and how graphs that express queries could be recorded textually. The programs written in this language could be used in distributed computing.

Chapter 4 describes the incremental development process and deliverables of the translation system that bridge Graph User Interface systems and the CORAL deductive system. Cooperating with the Transferable Graph Language(TGL), the translation system supports diagrammatic queries.

Chapter 5 concludes the thesis with an overview of the thesis and future work.

Appendix A presents database schemes of the university data model and table records(or facts) used to test the translation system.

Appendix B provides diagram examples of a whole set of queries in university data model.

Chapter 2

Background

2.1 Object-Oriented Incremental Process Model

The Figure 1 illustrates the incremental process model [22]. It provides a practical and useful alternative to deal with the situation of requirements increasing during development.

The entire development is broken down into many increments. In every increment, the waterfall approach is applied. When new requirements are identified, the current process activities will not be affected. Instead, the current increment will be completed. After that, the newly identified requirements are collected into another increment, and a new cycle of waterfall process begins. Only a mininmal increase of time should be expected to complete all the increments compared to finishing it in a single pass.

The whole process is manageable because every increment is a smaller but complete product compared with the whole application. This also yields high productivity because relatively fewer professionals are necessary to develop the smaller products. It ensures that the deliverables of all increments are robust by applying the waterfall model to each of them. The early increments help to understand better what to expect from the final product. This is useful when the application is complex and has a long schedule to complete. The functionality provided by the early products within schedules are enough to satisfy the clients, and thereby reduce the pressure on the developer team.

If the new requirements identified have a qualitative difference with those in complete increments , it is not enough any more to merely design new classes to integrate into the original design. The system structure achieved from early increments may need to be adjusted too. Consequently, more time will be required.

2.2 Graphlog and Hygraph

Graphlog [3] and Hy+ [5] were developed at University of Toronto under the leader of Mariano P. Consens. Graphlog is a language to assist in visualizing queries and Hy+ is the system using Graphlog to pose queries.The Hy+ system is implemented using the Object-Oriented language Smalltalk.

2.2.1 Fundamental Elements

2.2.1.1 Terms

In Graphlog, a *term* is either a constant, a variable, an anonymous variable, an aggregation function, or a functor. A *constant* is a number literal or a string literal. A *variable* is a sequence of characters in which the first character is a uppercase letter and the remaining characters are in the set of alphabetical letters, digits, '_', and '-'. However, Graphlog is case sensitive. For example, the words "import" and "Import" have different semantics. More likely, "import" is a predicate name, whereas, "Import" is a variable name. An *anonymous variable* in Graphlog is similar to that in Prolog [2]. Generally, an anonymous variable is represented by the symbol '_'. The aggregate functions defined in Graplog are the set {**MAX, MIN, COUNT, SUM, AVG**}. Their only permitted argument is a variable, which should represent numbers. A *functor* in Graphlog has the same semantics in logic, and is applied to a number of terms.

2.2.1.2 Expressions

Graphlog is the logic behind hygraph [3], and hygraph instances are presented in Hy+ [5] GUI system. Hy+ GUI system has two types of windows: defineGraphlog and showGraphlog.

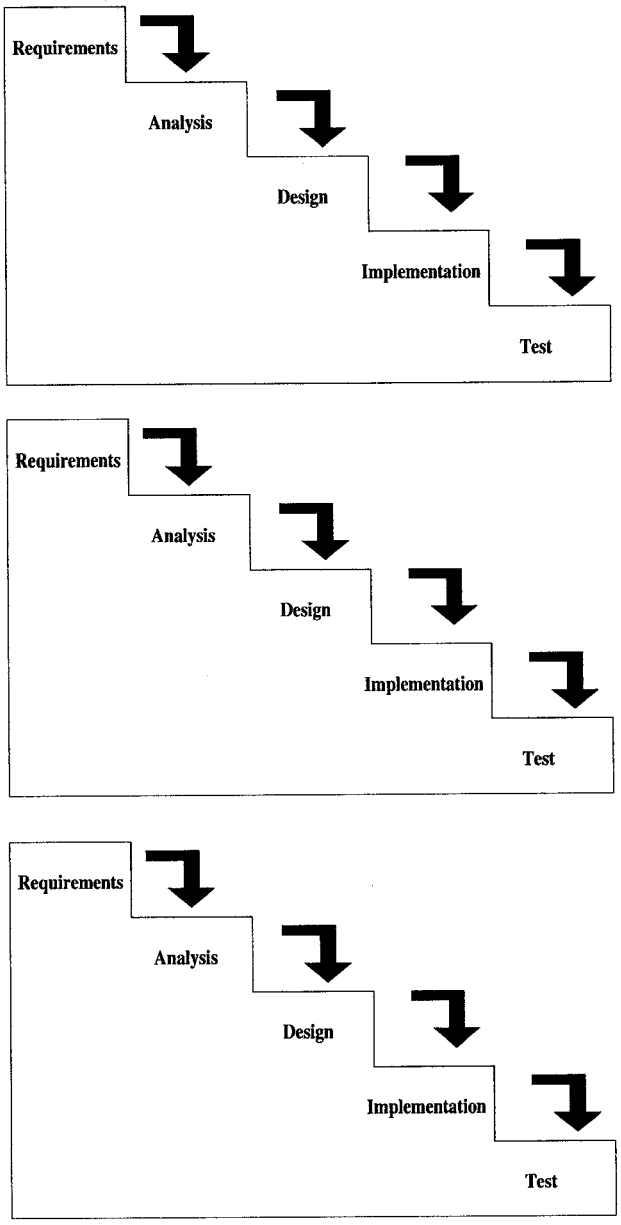


Figure 1: The Incremental Process Model

Graphlog expressions are visualized by applying hygraph patterns in Hy+ system. Hygraph is a hybrid of Harel's highgraph and directed hypergraphs [3]. It is depicted in the environment of Hy+ system. The elements of hygraph are node, directed edge, and blob. A *node* is a vertex in a graph, and a *directed edge* is an arrowed arc in a graph. A *blob* has the look of a rectangle box with a container node at upper left corner outside the box and a set of contained nodes distributed inside the box. In this way, the blob represents a relation between the container node and the contained nodes. So, in effect, a blob is a collection of the edges and it gathers nodes together. A blob can have two views, one is zoom-out, the other is zoom-in [3]. With the zoom-out view, visually a blob shows only the container node. While with the zoom-in view, a blob shows all its components described above. A blob can switch from a view to the other view dependent on the user's interest. Thereby, a zoom-out view provides more general picture as a whole, whereas a zoom-in view presents details.

The two types of hy+ windows correspond to two types of hygraph patterns. The first, a defineGraphlog is represented as a define query pattern of hygraph. Define-Graphog defines a new relation based on the rest context. Its corresponding define query pattern demonstrates the definition in a distinct way, by comparing with its context. Generally, the edge or blob outline that presents the new relation is distinguished.

In hygraph patterns, a node represents an entity, and is labeled by a Graphlog term. An edge between two nodes illustrates the relationship that involves them. The label of an edge or a blob is aligned near the edge or inside the blob to represent the relation. The *label* of an edge or a blob is a path regular expression generated by the following grammar, where $\bar{\mathrm{T}}$ is a sequence of terms and p is a predicate: $E \leftarrow E \mid E; E \cdot E; -E; \neg E; (E); E+; E*;\ \mathrm{p}(\bar{\mathrm{T}})$

Figure 2 is a defineGraphlog window. It is demonstrated through a define query pattern of hygraph. The new relation, *student_by_staff*, is between the entity student and the entity staff. Relevant information also includes another entity course, the relationship defined between the entity student and the entity course with the predicate *takes*, and the predicate *teaches* between the entity staff and the entity course.

The other showGraphlog window corresponds to filter query pattern of hygraph. The purpose of a filter query is to retrieve all the database instances that match the pattern. There are two different aspects between define query pattern and filter query

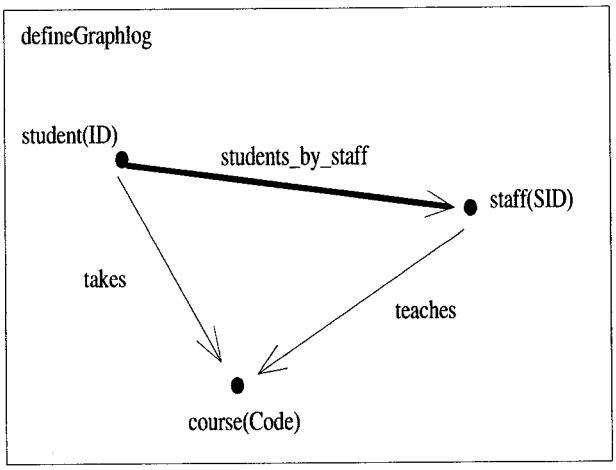


Figure 2: An Example of DefineGraphlog

pattern:

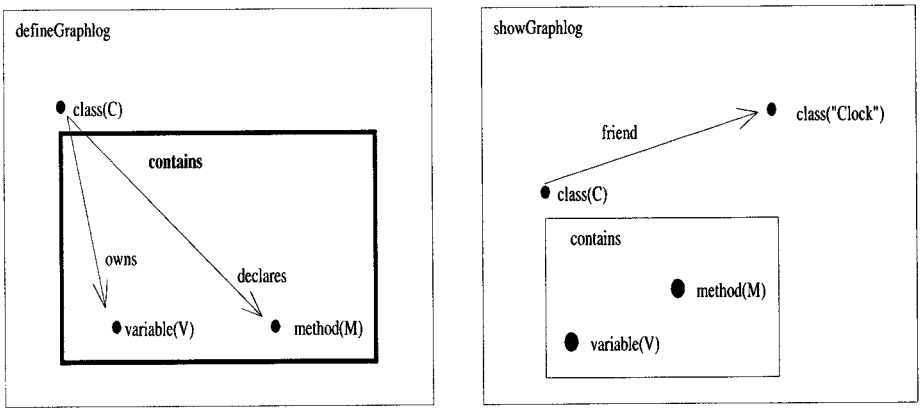
1. Distinguished objects have different meaning. In define query pattern, the distinguishment means defining a new relation, while in filter query pattern, it means retrieving existing data.

2. The number of distinguished objects permitted. There could only be one distinguished object in a define query, whereas there could be one to many in a filter query.

Additionally, there is difference between defining a blob and using a blob. Figure 3 gives examples respectively.The example within the box marked *defineGraphlog*, the relation *contains* is defined by the blob. It exists between a class and its variables or its methods. In the other example, the newly defined relation is employed. The purpose of this query is to retrieve all the members(variables and methods) of the classes that satisfies the relation *friend* with class *Clock*.

2.2.2 Aggregate Functions

Graphlog also supports aggregation on collected multiset of tuples. Graphlog provides five unary aggregate operators: MAX, MIN, COUNT, SUM, and AVG. They are applicable to the labels of distinguished objects in the defineGraphlog window.



An Example of Define a Blob

An Example of Using a Blob

Figure 3: Examples of Defining and Using Blobs

2.2.3 Graphlog, SQL and Object-Oriented Query Language

2.2.3.1 Similarity

The entity of Graphlog is similar to the relation in relational model or class in Object-Oriented model. The predicate label of an edge or a blob can be compared with the relationship in an E-R model and links in Object-Oriented model. Because a term can contain functors, the links within objects can be related easily. For example, a person who has the attribute of address is an object in Object-oriented model. The attribute address will be stored as another entity separately, and a reference is used to link a person entity and his/her address entity. The attributes of an address entity could be: Street, City, PostCode. Then the term in Graphlog that corresponds to the person entity in a Object-Oriented model is: Person(Name, Sex, Birthday, address(Street, City, PostCode)).

2.2.3.2 Power of Graphlog

Although the queries posed using SQL and Object-Oriented query languages can be expressed using Graphlog, a large set of Graphlog expressions can not be expressed using SQL or OO query languages. Graphlog is more expressive in the following aspects:

- Expressing recursive queries.
  Most current commercial query systems conform to SQL 92 that does not support recursive queries. DB2 supports SQL99 which has linear recursion.

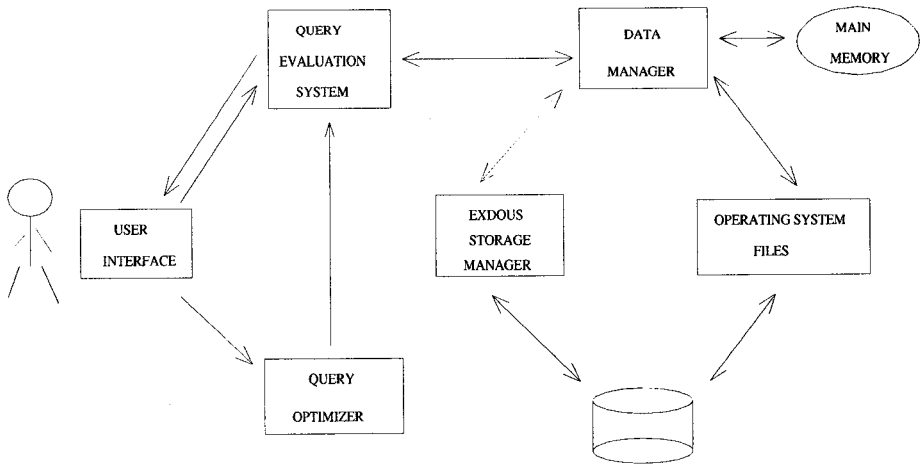


Figure 4: The Architecture of CORAL System

- More comprehensible.
  The queries in Graphlog are more comprehensible because of the diagrammatic depiction.

- Complex aggregation expressions.
  With the ability to collect multisets of tuples and to compute aggregate functions on them, Graphlog can pose a large number of queries that can not be expressed using relational algebra or relational calculus [4].

2.3 CORAL system

The CORAL deductive database system [6] is developed at University of Wisconsin, at Madison. The main contributors of this project are Raghu Ramakrishnan, Devesh Srivastava, S. Sudarshan, and Proveen Seshadri. The research began in 1988, and the current release version is 1.5.2, released on November 26, 1997. Being in the public domain, CORAL system is available at the web site: http://www.cs.wisc.edu/coral/.

2.3.1 Architecture of CORAL Deductive System

The architecture of the CORAL deductive system is shown in Figure 4.

CORAL is a single-user database system. Through its command-line interface, users can type in simple queries or consult programs to process complex queries. The query processing includes two main parts: a query optimizer and a query evaluation

system. Query optimizer will rewrite the complex queries that are defined in declarative "program modules". Then the query evaluation system interprets the internal form of the optimized annotated program, under the directives of annotations. Differing from other deductive systems such as Prolog system and LDL, which compile programs, CORAL system provides significantly high execution speed. Thus it is particularly suitable for interactive program development [6]. Using a "get-next-tuple" interface, the query evaluation system access to relations through the Data Manager system.

The EXDOUS storage manager has a client-server architecture. Thus, the CORAL system could extend its stand-alone mode to share data with other systems via the EXODUS storage manager. Persistent data has two storage formats: text files, which are strictly local, and the EXODUS storage manager. Data in text files will be converted into main memory relations, while data in the EXODUS storage manager is paged into EXODUS buffers on demand.

CORAL supports a bi-directional interface to C++:

1. CORAL code can be embedded in C++ code.

2. C++ can manipulate the database and C++ can define predicates used by CORAL.

2.3.2 CORAL Declarative Language

The declarative language provided by CORAL [7] can be used to express complex queries or view definitions on the database. CORAL combines features of database query languages, such as efficient treatment of large relations, aggregate operations and declarative semantics, with features of a logic programming language, such as powerful inference capabilities and support for structural and incomplete data. The CORAL declarative language widely extends the expressiveness of standard database query language such as SQL. In addition, CORAL differs from logic programming languages such as Prolog in supporting a declarative semantics.

The concepts of *constant, variable, term, fact*, and *rules* follow the convention of logic programming languages. We provide three key concepts as follows:

1. *Extensional Database(EDB)*
   A set of facts.

2. *Intensional Database(IDB)*

   A collection of rules. At compile time, only the IDB and meta-information about EDB are examined as programs.

3. *Datalog Program*

   Programs with only constants and variables as terms and without funtors, negation, set-grouping, aggregation, and built-in predicates.

2.3.2.1 Syntax and Semantics

Constants — Numbers, identifiers beginning with lower-case letters and quoted strings.

Variables — Identifiers beginning with an upper-case letter.

Functors — Uninterpreted function symbols which are represented using identifiers beginning with a lower-case letter.

Simple terms — Have the format of constants(constants/variables, ...).

Complex terms — Have the format of constants(constants/variables/functors(...), ...).

Rules — $p(\bar{t}) : -p_1(\bar{t_1}), \ldots, p_n(\bar{t_n})$. If $p_1(\bar{t_1}), \ldots, p_n(\bar{t_n})$ are true, then $p(\bar{t})$ is true. Rule is an assertion for all assignments of terms to the variables in the rule. Especially, a fact is a rule with an empty body.

Modules — Sets of rules and facts constitute *modules*. Modules export part or whole of the predicates they define and the query form of the exported predicates. "b" denotes an argument that must be bound in the query, whereas "f" is an argument that should be free. The exported predicates are visible to all other modules.

Non-ground facts — Non-ground facts are facts with variable arguments. The semantics is that the fact is true with any replacement of the variables. Such facts can be useful in knowledge representation and natural language process.

Negation — CORAL supports non-floundering left-to-right modularly stratified programs [7]. Non-floundering means all variables in a negative literal are bound before the literal is evaluated. Modularly stratified

denotes there are no cycles through negation during generation of the answers and sub-queries. The keyword "**not**" is the prefix indicating a negation body literal. Note that no negative head is allowed.

Sets and Multisets — In addition to terms, sets and multisets are values in CORAL. Sets and multisets can in turn contain sets and multisets. So the universe of discourse is an extended Herbrand universe, which is different from the standard logic programming. To obtain a set from a multiset, we can use the *makeset* operator. As with negation, the set-grouping operator is applied left-to-right modularly stratified.

Aggregate Operations — COUNT, AVG, MIN, MAX, and SUM are permitted aggregate operations. Although there is no clear group-by function, we can achieve the same effect by applying aggregate operation on grouped variables. For example, if we want to find the number of students in classes, given a register relation having two columns of *Class_No, Student_ID.* We can use the rule: *numberOfStu(Class_No,* $COUNT(<$ *Student_ID* $>)$ *:- register(Class_No, Student_ID).* to define the predicate *numberOfStud*, then query the predicate to find the results.

2.3.3 The Query Optimization

A declarative language is not supposed to specify the evaluation strategy. However, CORAL has a series of annotations that do allow the program writer to decide the optimization strategy and obtain better performancec̃iteCoralLanguage.

2.4 Extensible Markup Language(XML)

XML [12] was approved by the World Wide Web Committee(W3C) on February 10, 1995. XML has been developed to make the exchange of data on the web easier and more efficient.

2.4.1 General Definitions

XML XML [14] is a meta language, that is, a language for describing languages. A meta language such as XML, is broader than metadata because it provides the syntax that allows users to create their own markup language and define their own vocabularies to meet specific application or industry needs. A markup language uses tags embedded directly into the text to specify or increase the meaning of the enclosed piece of text.

Valid XML XML documents that make use of internal or external Data Type Definition files are known as "valid" XML and require an XML parser to check incoming data against the rules defined in the DTD to verify that the data were structured correctly.

Well-formed XML An XML document is well-formed if it has a sound logical structure and syntax. XML only requires data to be "well-formed". A non-validating XML parser checks if an XML file is "well-formed" but does not check if it is valid.

XML Vocabularies XML vocabularies are those markup languages defined using XML technology. Especially, HTML can be a kind of XML vocabulary. While you can create your own markup, many XML vocabularies are developed for various fields and received recognition. Correspondingly, specialized tools and browsers will be developed for these vocabularies so that the performance is much higher and the implementation is much more adapted.

2.4.2 The relation of XML with SGML and HTML

While XML is a meta-markup language and HTML is a specialized markup language. Standard Generalized Markup Language(SGML) is the basis for all markup languages, including XML and HTML.

XML is a subset of SGML [14]. SGML is a text processing standard that describes how a document should be laid out and structured. As a dialect of SGML, XML describes the information content of a document. SGML documents use a Document Type Definition(DTD) to specify the structure of a document. Sometimes, the terminology XML Data Schema is used to distinguish from SGML DTDs. Because SGML

	XML	HTML	SGML
Document reusable	Yes	No	Yes
User-Defined tags	Allowed	Not Allowed	Allowed
Tag number	Unlimited	Fixed in each version	Unlimited
Information-oriented	Yes	No	Yes
Processing directions	Yes	No	Yes
Complexity	Medium	Simple	Overkill
DTD Type	Could have	Never have	Have

Table 1: A Comparison of XML, SGML and HTML

is complicated to learn and apply, people focus on using the HyperText Markup Language(HTML), which, in effect, in its pure form, is an application of SGML with a Document Type Definition. However, HTML is more presentation-oriented. It does not handle the meaning of the information displayed. Additionally, HTML has a fixed set of tags, although it is extended with versions. In contrast, XML lets users define their own tags which are much more flexible and vendor independent. Further, an XML document can be developed once, but used many times.

XML was designed to be easily implemented and to work with both SGML and HTML. In the foreseeable future, XML and HTML will coexist in the web world and HTML will be a necessary part of implementing XML solutions on the web.

Table 1 lists the similarities and difference among XML, SGML and HTML.

2.4.3 XML Constructs

The main constructs of XML documents are elements, attributes, and entities. The following are the main syntax for XML document constructs:

- Elements: < ! ELEMENT element-name EMPTY (#PCDATA) (organization of subelements) >
- Attributes: < ! ATTLIST element-name attribute-name attribute-type if-required >
- Entities: < ! ENTITY entity-name "content" >

- Processing instructions: < ? name instruction ? >. E.g., < ? XML version="1.0" >

Elements — Fundamental structures for an XML document. They are containers for XML document content, and an XML document is constructed by such containers. Elements are defined in DTDs and are represented by tags in the documents. Also, the elements definition specifies the content model of XML, that is, how elements are nested.

Attributes — Modifying elements, but not the XML document contents. They provide adornment information for elements. Although most HTML attributes mainly function to format the content with the purpose of separating display from content, XML DTDs seldom have formatting attributes. Generally, style sheets are responsible for representation.

Entity — A chunk of data that is defined in a DTD before being referred. It can be textual data, binary files, parameters, and characters.

Processing instruction — Directing how a document will be processed with a XML processor.

DTD — Regulating how XML documents are built. First, DTDs specify the XML constructs for XML documents. Next, DTDs regulate how the constructs are connected. In addition, DTDs will provide instructions for processing the XML documents. There are two types of DTDs: external DTDs and internal DTDs. External DTDs make general declarations, while internal DTDs make document specific declarations.

2.4.4 XML Processors

The basic functions of an XML processor include reading documents, interpreting their markup and acting as instructed.

While most currently available processors are just parsers that are not responsible for displaying document, there are some processors that both parse and display documents.

A validating processor checks a document against the DTD to make sure that the document strictly adheres to the rules set down in the DTD. If the document violates any part of the DTD, the parser will return an error, and the process of the document may cease entirely. This is not necessarily a bad thing. You can use the validating processor to check your documents for accuracy.

2.5 Java

Java [21], developed at Sun Microsystems, is an object-oriented programming language based on proven technologies. Java has been extensively applied in network related systems since its invention. This mainly lies in its following key features :

- Portability. Compiled Java programs can run across different platforms, given that platform has Java environment, and the functionality remains the same. Thus, in network applications, different platforms will not require additional efforts to execute Java programs.

- Pure Object-Oriented. Java is a pure object-oriented programming language. Dangling procedure functions are eliminated, and all the method calls are proceeded through messaging receiver objects from client objects. Thereby consistent structures are achieved.

- Distributed. Java provides extensive frameworks to support network communications.

2.5.1 The Architecture of Java

The Java Platform The Java application programming interfaces (APIs) and the Java1 virtual machine (JVM) together compose the Java platform. Java virtual machine provides the environment that Java programs can be run (or interpreted).Its appropriate version is installed in a computer system depending on the platform the computer system runs. Then the Java virtual machine compiles or interprets java programs before it transfers them to the underlying operating system. So Java programs are platform independent [17].

Security Java protects the safety in the following key aspects for computer systems that run Java programs [19] :

- Shielding Memory Access
  There is no pointer arithmetic in Java programming language. Thus, it eliminates the run-time crashes caused by inappropriate access to memory areas.

- Automatic Garbage Collection
  JVM provides Garbage Collection mechanism to release no longer referenced memory without the intervence of programmers. Because all the dynamic memory are allocated from the heap, the garbage collection only monitors references to heap. If there is no references to an area of it, the area will be freed for reallocation.

- Verifying Byte-code
  Before executing byte-code, the JVM will verify the byte-code to ensure it is the output of a legitimate Java compiler.

- Applet Security Manager
  Through SecurityManager subclasses, browsers ensure no potentially unsafe operations are performed through an Applet. These operations include: reading/writing local file system, opening sockets to destinations other than the IP address from which the applet originated, creating a ServerSocket for listening on a port, and reading some of system properties.

Applications and Applets Java programming language provides two structures of programs: applications and applets. The key difference of the two types of programs lies in that an application can fully access system resources, whereas an applet is embedded in web pages and has restricted access to system resources. In addition, the methods of invoking them are different. An application is invoked by a command "java -option classname", whereas an applet is invoked automatically whenever its embedding web page is loaded.

2.5.2 Collections Framework [20]

A collection is a container that groups other objects, and these objects are called its elements. Further, the elements of the collection can be stored, retrieved, and

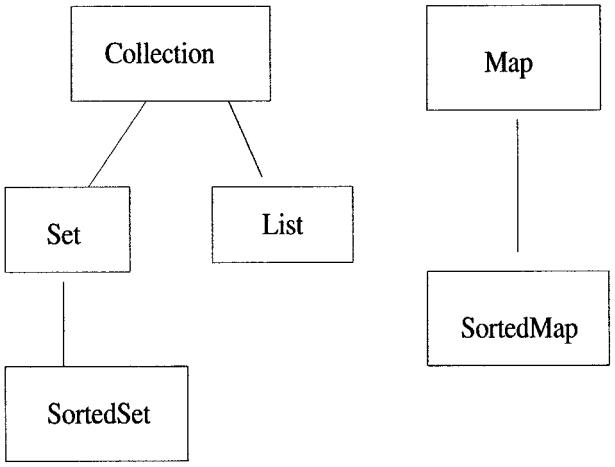


Figure 5: Core Collections Interface Hierarchy of Java

manipulated through it. Collections are abstracted from reality. Examples are file folders, binary trees, and sets.

Java provides a collections framework to achieve the following goals [20]:

- Reducing Programming Efforts. It eliminates the needs to reinvent data structures and algorithms in each program and lets the programmers focus on business logic. It also provides interface for collections to transform to each other so that data could be easily reused.

- Increasing Programming Speed and Quality. The algorithms and implementations of the collections framework are well-designed and of high quality. By conforming to the framework, first, programmers save time in designing the whole structure. Secondly, programmers can make use of the structure for consistent programming design, thereby achieve high quality.

The Java collections framework has three main parts: interface, implementation, and algorithm.

2.5.2.1 Interface

The interface is the core of Java collections framework. Figure 5 shows the core collections interface and the hierarchy.

Collection The Collection interface is the root of the collection hierarchy. Java does not provide implementation for it. Its purpose is to provide the generality of

all the subclasses so that subclasses can be specified dynamically instead of compile time.

Set The Set interface models the set concept. So it does not allow duplication of elements. However, it does not require ordered elements although they could be ordered.

SortedSet First, a SortedSet interface has the characteristics of a Set interface. Secondly, all the elements in it will be in ascending order. The order on the elements is the natural order or regulated according to a COMPARATOR provided together with the creation of SortedSet.

List List is an ordered Collection with potentially duplicate elements. List allows access to its elements at exact positions. That means it is possible to refer to an element by its index in the list.

Map The Map and its subclass SortedMap comprise the other collection hierarchy. The Map is the root. Map has such structures and capabilities that keys and their corresponding values can be matched in the Map. The Map does not allow duplication. A key can only have one copy in the Map. Moreover, only one value of the key can be stored in the Map.

SortedMap A SortedMap is a Map that maintains its entries in the ascending order of the keys. Similar to the SortedSet in Collection hierarchy, the order of the SortedMap follows the natural order of keys or computed according to the associated Comparator.

Although Map and Collection are in different collection hierarchy, a Map could be viewed as a Collection in three ways:

1. KeySet: The keys contained in the Map comprise a Set.
2. Values: The values that match the keys is in effect a Collection. However, This Collection is not necessarily a Set because the same values can be associated with different keys.
3. entrySet: All the key-value pairs in the Map together constructs a Set.

2.5.2.2 Implementation

The implementation of collection interface provides concrete classes that inherit the features of the interface.

The implementations are divided into three groups according to the functions achieved:

General Purpose Implementation Java provides a series of public classes for the primary implementation of the core collection interface.

Wrapper Implementation Wrapper Implementation complies to the Decorator Design Pattern [17]. It allows implementations to have added functionality of others. These implementations are in Collections API.

Convenience Implementation The convenience implementations are realized using static factory methods or exported constants. The examples are: Arrays.asList method, Collections.nCopies method, Collections.singleton method, and constants of Collections.EMPTY_SET plus Collections.EMPTY_LIST.

2.5.2.3 Algorithm

The algorithms come from Collections API. Their purpose is: sorting, shuffling, routining data manipulation, searching, and finding extreme values.

2.5.3 The Input/Output Framework

Java provides an Input/Output framework to support flexible and simple configuration. There are two types of I/O streams in Java Programming language: byte streams and character streams. Byte streams support reading and writing any type of data, including strings and binary format, whereas character streams support reading and writing text depending on the locale character encodings.

2.5.3.1 Character Streams

Reader and Writer are the abstract super classes of a set of text reading classes. They provide the common API and partial implementation. Figure 6 shows the two sets of hierarchies supporting reading and writing text.

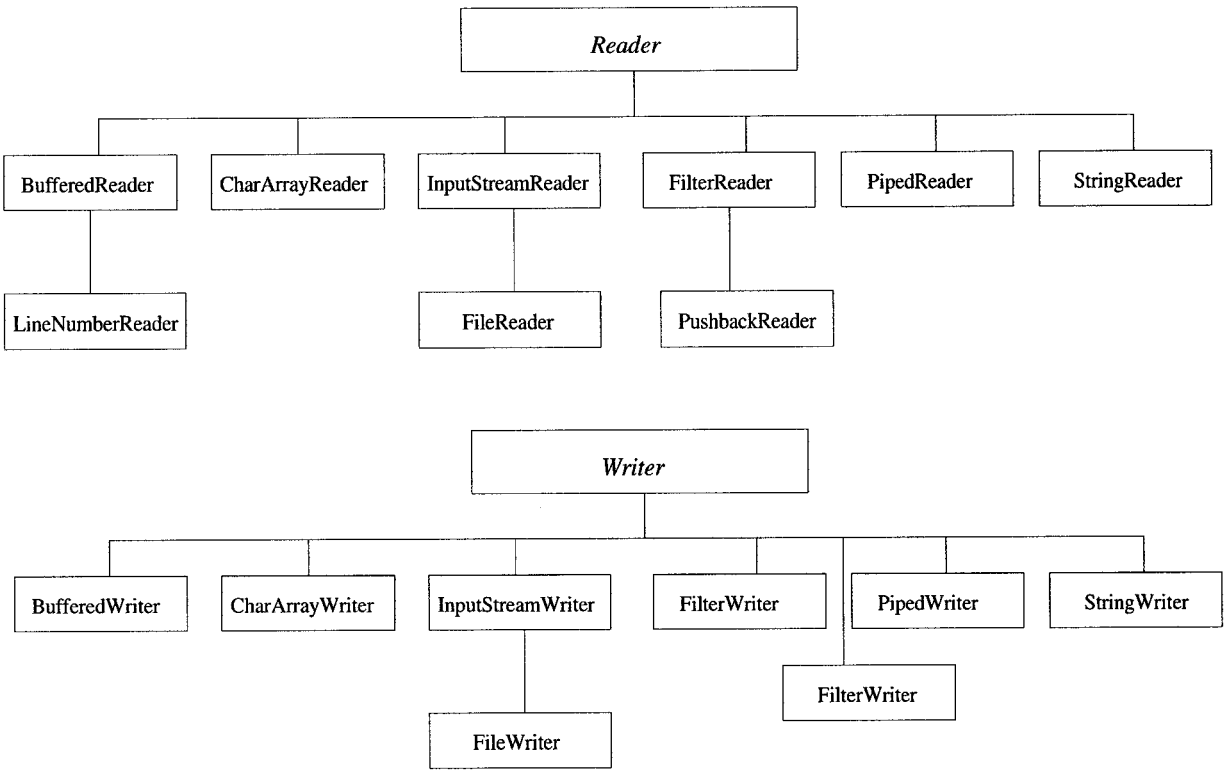


Figure 6: Character Stream Class Hierarchy of Java

Character streams are responsible to read and write 16-bit characters.

2.5.3.2 Byte Streams

Byte streams read or write 8-bit bytes, and all the byte streams classes are descendants of InputStream and OutputStream. They inherit the partial implementation from InputStream and OutputStream. Binary data like images and sounds are typically read and written using the byte streams. The byte streams family also has two members, ObjectInputStream and ObjectOutputStream, which are used for object serialization.

Figure 7 illustrates the two categories byte streams that are similar to those in Figure 6.

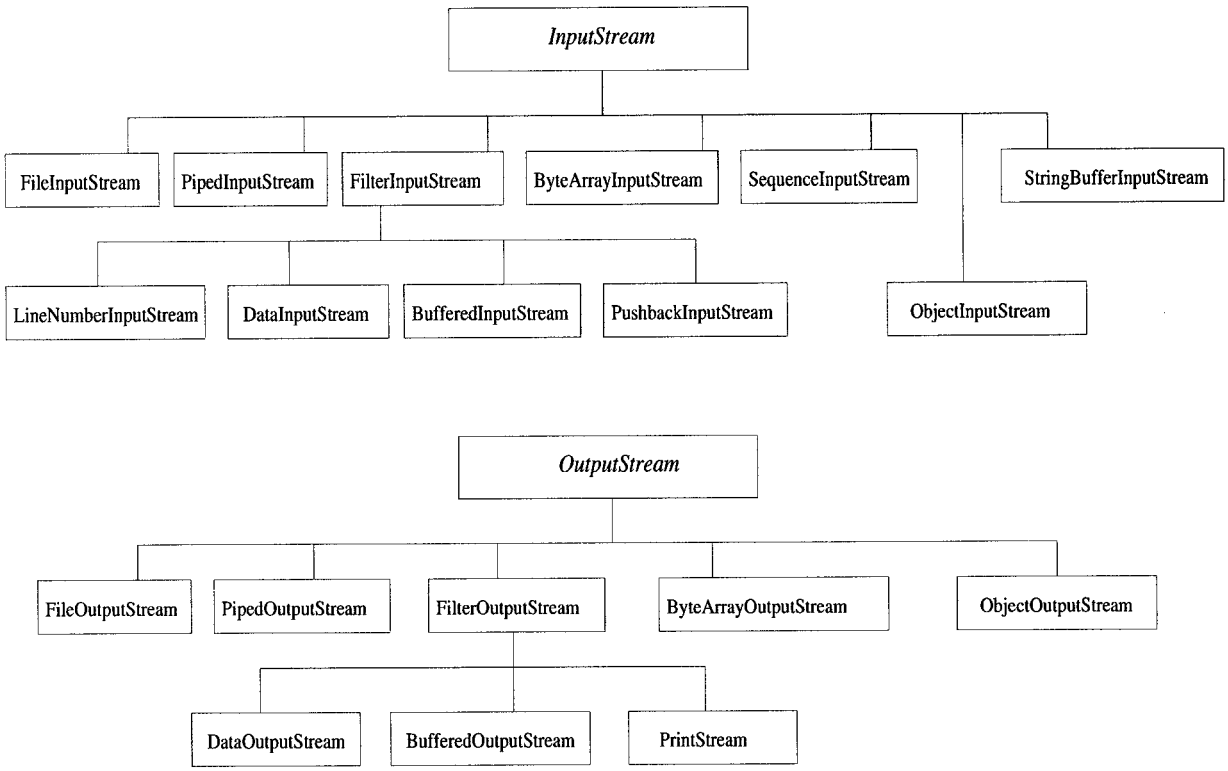


Figure 7: Byte Stream Class Hierarchy of Java

Chapter 3

Transferable Graph Language

This chapter covers the syntax and semantics of the Transferable Graph Language, which is abbreviated as TGL. It describes all aspects of the language, including the lexical elements, semantics of all types, as well as program structure.

The Transferable Graph Language is a platform-independent text language that describes the information of query graphs, especially it emphasizes the data expressed instead of the visualization effects. It is designed to be shared by many applications and exchanged in a network environment. The graphs depicted in this language consists of labeled nodes, directed edges, and blobs. A blob is a box that associates an outer node with a set of inner nodes. Essentially, a graphs depicted by TGL characteristically resembles that in a hygraph pattern. However, Transferable Graph Language has a family of built-in predicates while Graphlog does not have this feature, so the graph instances supported by TGL are a superset of those in hygraph patterns because the TGL allows more concise and expressive expressions than Graphlog does.

The rationale for inventing such a language lies in the need for passing diagrammatic queries for analysis and recording the corresponding diagrams. It is expensive and impractical to send a whole diagram to analysis applications. Instead, the interested parts and associated relationships could be abstracted from the diagram and packed as messages using a text language. Additionally, because the process of such data may be distributed to several computers in a distributed system and the query results will probably be utilized by many applications, it is better that the text language has the feature of portability. This is achieved by designing TGL program

structure as XML Data Type Definition(DTD) and thereby enforcing the TGL programs to follow XML patterns. The design of taking on XML DTDs has the following benefits:

- Reduce Design Errors. Since the XML DTD syntax has been well-established, errors can be reduced in developing the language.
- A program written to conform to XML syntax has high readability for human being so that errors are easy to identify.
- The most important thing is that the program will be suitable for transferring across a network and reusing in multiple platforms.

3.1 Lexical Elements

This section presents the elements of the Transferable Graph Language.

3.1.1 Characters used

The Transferable Graph Language adopts the Unicode Character Set instead of ASCII Character Set. Unicode Character Set is the international standard of 16-bit characters. It supports all the special logic operators used in Transferable Graph Language, such as ' ¬ ' (not). And other non-ASCII characters in the Unicode Character Set can only appear in string literals.

3.1.2 Identifiers

Transferable Graph Language identifiers comprise two types. One type is for node names, edge predicates, and blob predictates, the other for field names of a node and attribute names of predicates.

- Predicate Identifiers
  They begin with a lower-case letter followed by letters, the underscore('_'), the dollar sign ($), and digits as well.
- Attribute Identifiers
  They begin with an upper-case letter followed by letters, the dollar sign ($) and

digits. An exception is a single underscore character ('_') can represents a field, meaning the field is anonymous.

3.1.3 Primitive Types and Literals

Transferable Graph Language provides the following primitive types:

- Integer
- Floating-point
- String

The constant values of each types are expressed as literals.

Integer Type Transferable Graph Language allows positive integer type and the maximum value is 9223372036854775807. Integer literals are written in numbers. An integer literal starts with a nonzero decimal digit, followed by decimal digits. For instance, 30.

Floating-point Type Floating-point literals are written as a decimal number. Some examples of floating-point literals are 0.3, 3.145, and 563.2.

String Type Transferable Graph Language permits string type but not character type. All characters can be represented using string type by having only one character. A string literal is enclosed in double quotes and consists of a sequence of characters, as in the following examples: "Saint George", "m", "Air".

3.1.4 Operators

Transferable Graph Language provides two categories of operators: one is arithmetic oriented; the other is logic-related. The built-in predicates belong to the second category. Operators are used for constructing edge or blob labels and node values.

3.1.4.1 Arithmetic Operators

The infix binary arithmetic operators list in Table 2 can only be applied to node label, but not to edge or blob labels.

*	Multiplication
/	Division
+	Addition
−	Substraction

Table 2: The Arithmetic Operators Applicable to TGL

3.1.4.2 Assignment Operator

The assignment operator is the simple '='. The operator is only for constructing an edge label. For instance, an edge label could be '='.

3.1.4.3 Comparison Operators

The comparison operators include equality operator ==, inequality operator <>, less than < , less than or equal to <=, greater than > , and greater than or equal to >=. These are applicable to edge labels. Note that a symbol '&' is required to prefix the operators <>, <, <=, > *and* >= . because the construction of programs follows XML shemas, where the symbol ' <' is used to mark the beginning of a tag,

3.1.4.4 Conjunction Operators

Transferable Graph Language bears more powerful expressiveness by providing operators to connect expressions. The newly formed expressions conjoined by the conjunction operators lead to concise queries. The operators and their semantics are shown in Table 3. The precedence associated with each operator and the position where the operator sits in expressions are listed along with the operator as well. The priority for the precedences decreases from lower to higher numbers. Note that it is not inputable from the keyboard for the symbol ¬, so the character reference '&#00AC;' is used to represent the ¬ in the program in accordance to XML.

3.1.4.5 built-in operators

is_a Operator The *is_a* operator is a label for a directed edge. The semantics is that the pointing entity of the edge has a generation relationship with the pointed node. Put in another way, the key of the from-node entity is the same as that of the

Conjunction Operator	Semantics	Position	Precedence
()	exlusively evaluated first	enclosing	1
¬	not	prefix	2
-	inverse	prefix	3
+	transitive closure	suffix	4
*	transitive closure, including self	suffix	4
|	or	infix	5
·	concatenation	infix	5

Table 3: The conjunction Operators Applicable to TGL

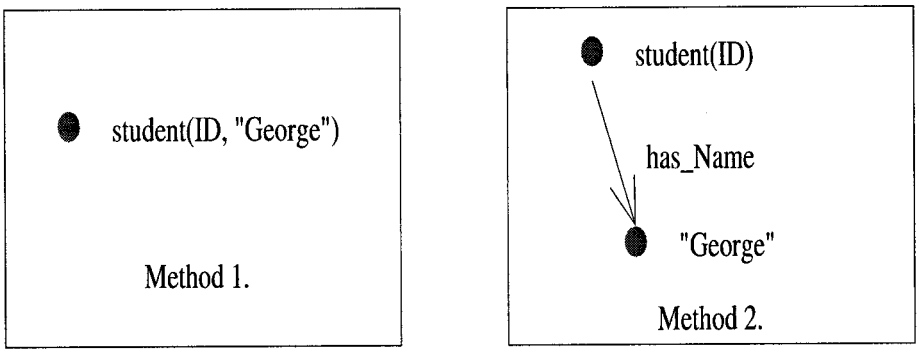


Figure 8: The Two Methods to Depict Attributes

to-node although the two nodes are different relations. The *is_a* operator indicates the two nodes are implicitly related.

has_XXX Operators This kind of operators provide another way to present an attribute of an entity by using an edge. Generally, the pointing entity does not list all attributes explicitly and the pointed entity is the value of a specific attribute. The "XXX" in the format of *has_XXX* will be replaced by the specific field name. For example, the Figure 8 provides two methods to depict the tuples with the relation scheme of student(ID, Name) and the Name attribute having the value of "George".

3.1.5 Expressions

There are two types of expressions in Transferable Graph Language: arithmetic expressions and logical expressions. Arithmetic expressions can be applied as node labels, while logical expressions act as edge or blob labels.

$E ::= E \ , E \mid E . E \mid E + \mid E * \mid - E \mid \neg E \mid (E) \mid$ *Compound – predicate*

Compoound–predicate ::= predicate(simple–attribute–list)
| predicate(attribute–list–with–functor)

simple–attribute–list ::= simple–attribute, simple–attribute–list

attribute–list–with–functors ::= attribute, attribute–list–with–functors

attribute ::= simple–attribute | functor–attribute

simple–attribute ::= identifier | literals

functor–attribute ::= Compound–predicate

Figure 9: TGL Logical Expression Syntax

3.1.5.1 Arithmetic Expressions

Arithmetic expressions are generated by the following grammar:

$$
\begin{aligned}
E &::= E \ op \ E \\
&\mid (E) \\
&\mid N \\
N &::= Variable \\
&\mid Decimal\ literal \\
op &::= + \mid - \mid *
\end{aligned}
$$

Note that the parentheses have the highest priority and can change the ordinary precedences of ops.

3.1.5.2 Logical Expressions

The logical expressions follow Graphlog Path Regular Expression, but provide further regulation and description. The logical expressions serve as edge or blob labels. The grammar is as in Figure 9.

Here the predicate has the same meaning as in other logical languages [2]. From the definition, it can be seen that compound predicates could be nested.

The interpretation of an expression involves the nodes at the both ends of the edge or the nodes as container and containees of the blob.

Suppose A1 and A2 are the key attributes of the nodes incident to the edge or the blob with the expression, and E is the label of the edge or the blob. Let the final predicate derived for the edge or blob be represented as $P_e(A1, A2, \bar{T})$. The $\bar{T}$ is a vector of attributes. Then the program produced to derive the final predicate will contain the following rules:

- if $E ::= aPredicate(\bar{T})$, then $P_e(A1, A2, \bar{T}) :- aPredicate(A1, A2, \bar{T})$.
- if $E ::= \neg E1, and\ E1 ::= aPredicate(\bar{T})$, then $P_e(A1, A2, \bar{T}) :- \neg aPredicate(A1, A2, \bar{T})$.
- if $E ::= -E1, and\ E1 ::= aPredicate(\bar{T})$, then $P_e(A1, A2, \bar{T}) :- aPredicate(A2, A1, \bar{T})$.
- if $E ::= E1+, and\ E1 ::= aPredicate(\bar{T})$, then $P_e(A1, A2, \bar{T}) :- aPredicate(A1, A2, \bar{T})$. $P_e(A1, A2, \bar{T}) :- aPredicate(A1, X, \bar{T}), P_e(X, A2, \bar{T})$, where X is a variable.
- if $E ::= E1*, and\ E1 ::= aPredicate(\bar{T})$, then $P_e(A1, A1, \bar{T}) :- aPredicate(A1, A2, \bar{T})$. then $P_e(A2, A2, \bar{T}) :- aPredicate(A1, A2, \bar{T})$. $P_e(A1, A2, \bar{T}) :- aPredicate(A1, X, \bar{T})$, $P_e(X, A2, \bar{T})$, where X is a variable.
- if $E ::= E1|E2, and\ E1 ::= aPredicate1(\bar{T_1}), E2 ::= aPredicate2(\bar{T_2})$, then $P_e(A1, A2, \bar{T}) :- aPredicate1(A1, A2, \bar{T_1})$. $P_e(A1, A2, \bar{T}) :- aPredicate2(A1, A2, \bar{T_2})$.
- if $E ::= E1.E2, and\ E1 ::= aPredicate1(\bar{T_1}), E2 ::= aPredicate2(\bar{T_2})$, then $P_e(A1, A2, \bar{T}) :- aPredicate1(A1, X, \bar{T_1}), aPredicate2(X, A2, \bar{T_2})$.

3.2 Query Program Structure

TGL query program structure is stipulated through XML data definition type, graphlog.dtd, listed in Figure 10.

3.2.1 Program Elements

A TGL program may contain these elements:

- graphlog — describes a set of GUI patterns for a specific query
- defineGraphlog — identifies a pattern used to define a new relation or predicate

```
<!ELEMENT graphlog((defineGraphlog+, showGraphlog*)|
        (defineGraphlog*, showGraphlog+))>
<!ELEMENT defineGraphlog(include*, distinguished-define, content)>
<!ELEMENT showGraphlog(include*, ID, distinguished-show, content)>
<!ELEMENT distinguished-define(node|edge|blob)>
<!ELEMENT distinguished-show((node+,edge*,blob*)|
        (node*,edge+,blob*)|
   (node*,edge*,blob+))>
<!ELEMENT content(node*, edge*, blob*)>
<!ELEMENT node(ID, entity)>
<!ELEMENT entity(name, field*)>
<!ELEMENT edge(ID, predicate, fromNodeID, toNodeID)>
<!ELEMENT blob(ID, predicate, outerNodeID, innerNodeID+)>
<!ELEMENT ID(#PCDATA)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT field(#PCDATA)>
<!ELEMENT predicate(#PCDATA)>
<!ELEMENT fromNodeID(#PCDATA)>
<!ELEMENT toNodeID(#PCDATA)>
<!ELEMENT outerNodeID(#PCDATA)>
<!ELEMENT innerNodeID(#PCDATA)>
<!ELEMENT include(#PCDATA)>
```

Figure 10: The TGL Program Structure Regulation

- showGraphlog — identifies a pattern used to find the result interested
- distinguished-define — specifies the characteristics of the newly defined relation
- distinguished-show — describes the highlighted elements to query
- content — describes the context of a newly defined relation or queries made.
- node — describes the features of a node
- edge — specifies the constitution of an edge
- blob — defines the organization of a blob
- ID — identifies a showGraphlog, node, edge or blob in a particular graphlog.
- entity — provides the physical description of a node
- name — identifies the relation name of a node
- field — describe an attribute of a relation

- predicate — specifies the predicate name of an edge or a blob
- fromNodeID — specifies the node ID at the start point of an directed edge
- toNodeID — specifies the destination node of an directed edge
- outerNodeID — identifies the container node of a blob
- innerNodeID — identifies a contained node of a blob
- include — describes which predefined relation is involved

3.2.2 Content Models

A content model complements an element definition and regulates how elements can be nested given an XML vocabulary [15].

The Transferable Graph Language query program structure conforms to the element content model. Non-text element definitions and their semantics are listed in Table 4 and 5.

The query program structure declares the interface to the upper layer system. A GUI application that likely acts as the client will collect the drawing information and build up a TGL program according to the query program structure. Then the GUI application sends a message to the translation system with the TGL program for further treatment.

3.3 Result Structure

The query results derived by the CORAL system can not be utilized by the upper layer system directly. Because the links between drawing elements and results can be retained when forming CORAL programs, it is better to require the translation system to take care of transforming the results to a format for the upper layer system and other potential applications. Hence, Transferable Graph Language specifies the Result Structure for the process. The result structure is regulated using XML data definition type as well and is listed in Figure 11.

The result structure specifies the interface that the translation system should conform to when translating back the results. The upper layer system will also refer

Definition	$<!ELEMENT graphlog((defineGraphlog+, showGraphlog*)\|$ $(defineGraphlog*, showGraphlog+)) >$
Semantics	A graphlog may contain defineGraphlogs and showGraphlogs. However, at least one defineGraphlog or one showGraphlog.
Definition	$<!ELEMENT defineGraphlog(include*, distinguished-define,$ $content) >$
Semantics	A defineGraphlog employs zero to many predefined relations, specifies the features of the relation to be defined, and describes the context involved to define the new relation.
Definition	$<!ELEMENT showGraphlog(include*, ID, distinguished-show,$ $content) >$
Semantics	A showGraphlog uses zero to many predefined relations, provides an identification, which later acts as the clue for returned results, specifies teh list of elements to be queried, and provides the content.
Definition	$<!ELEMENT distinguished-define(node\|edge\|blob)$
Semantics	A distinguished-define element provides the interested node, edge, or blob descriptions. However, only one of them.
Definition	$<!ELEMENT distinguished-show((node+, edge*, blob*)\|$ $(node*, edge+, blob*)\|$ $(node*, edge*, blob+)) >$
Semantics	A distinguished-show element lists all highlighted drawing elements: nodes, edges, blobs. But at least one of them.
Definition	$<!ELEMENT content(node*, edge*, blob*) >$
Semantics	The content element describes all non-distinguished drawing elements. There may be many nodes, edges, and blobs are involved. Specially, there may not be any one.
Definition	$<!ELEMENT node(ID, entity) >$
Semantics	A node element specifies the identification in the graphlog and the physical features of a node.
Definition	$<!ELEMENT entity(name, field*) >$
Semantics	An entity element consists of a name and zero to many fields. It serves to detail a node information.

Table 4: The Content Model of TGL Query Program

Definition	$<!ELEMENT edge(ID, predicate, fromNodeID, toNodeID)>$
Semantics	An edge element details the identification, predicate name, the starting node identification , and the end node identification of an edge in a graphlog.
Definition	$<!ELEMENT blob(ID, predicate, outerNodeID, innerNodeID+)>$
Semantics	An blob element describes a blob in a graphlog. The data includes its identification, predicate name, the container node identification, and one to many inner node identifications.

Table 5: The Content Model of TGL Query Program(continue)

to the result structure to draw the diagram representing the query results for the human readers when it receives the translated results.

3.3.1 Program Elements

Accordingly, a translated result may include some or all of the following elements:

- showGraphlogReturn — describes the regulated results for queries in a showGraphlog
- result — details the values of the query
- node — specifies the associated values for interested fields of a node
- field — specifies value for a particular field in a node
- edge — provides the answers of from node and to node that constitute an edge
- fromNode — supplies the value of the starting node of an edge
- fromNode — supplies the value of the ending node of an edge
- blob — details the key values of the outer node and associated inner nodes for a blob
- outerNode – supplies the value of the container node of a blob
- innerNode – supplies the value of a contained node of a blob

```
<!ELEMENT showGraphlogReturn(result+)>
<!ATTLIST showGraphlogReturn ID CDATA>
<!ELEMENT result(node*, edge*, blob*)>
<!ELEMENT node(field+)>
<!ATTLIST node ID CDATA>
<!ELEMENT field(#PCDATA)>
<!ATTLIST field pos CDATA>
<!ELEMENT edge(fromNode, toNode)>
<!ATTLIST edge ID CDATA>
<!ELEMENT fromNode(#PCDATA)>
<!ATTLIST fromNode ID CDATA>
<!ELEMENT toNode(#PCDATA)>
<!ATTLIST toNode ID CDATA>
<!ELEMENT blob(outerNode, innerNode+)>
<!ATTLIST blob ID CDATA>
<!ELEMENT outerNode(#PCDATA)>
<!ATTLIST outerNode ID CDATA>
<!ELEMENT innerNode(#PCDATA)>
<!ATTLIST innerNode ID CDATA>
```

Figure 11: The TGL Result Structure Regulation

These elements also employs attributes for further description. And below the characteristics of these attributes are given:

- showGraphlogReturn-ID — provides the link between the query pattern and the result
- node-ID — specifies which node the value is related to
- field-pos — specifies the position of the field in the attribute list of a node
- edge-ID — provides the identification of the edge that values associated with
- fromNode-ID — specifies that the value links to the starting node of an edge
- toNode-ID — specifies that the value links to the ending node of an edge
- blob-ID — links a set of values with a blob throught the identification
- outerNode-ID — identifies the container node of a blob
- innerNode-ID — specifies a inner node identification of a blob

Definition	<!ELEMENT showGraphlogReturn(result+)>
Semantics	A showGraphlogReturn is constituted by one to many result units.
Definition	$<!ELEMENT result(node*, edge*, blob*)>$
Semantics	A result includes values of zero to many nodes, zero to many edges, and zero to many blobs.
Definition	$<!ELEMENT node(field+)>$
Semantics	A node value is given by the list of its field value.
Definition	$<!ELEMENT edge(fromNode, toNode)>$
Semantics	An edge value is described throught the from node and the to node
Definition	$<!ELEMENT blob(outerNode, innerNode+)>$
Semantics	A blob value is provided through the description of a set of nodes including a container node and associated contained node.

Table 6: TGL Result Content Model

3.3.2 Content Models

The content models for the result structure definition describe the relationships between elements, and the order of nested elements in their parent element. Details is given in the Table 6.

The elements field,fromNode,toNode, outerNode and innerNode are all text strings that provides concrete value.

3.4 Comments

A program without any annotation has low reusability and poor maintainability. Even the developers themselves may need to refer to the comments to analyze or modify the program. Comments are especially important for maintenance after the application launches.

Transferable Graph Language allows developers to use comments to provide text information of a program. A comment begins with the symbol - - and continues until

the end of that line. Through this function, TGL programs are capable of making a meaningful archive.

Chapter 4

Translation System

This chapter describes the process of the translation system. It follows the Object-Oriented incremental process [24] to collect requirements, analyze, design, implement, and test.

4.1 Context of The Translation System

Figure 12 depicts the context of the translation system. End users utilize the Diagrammatic Query System to define views and , more importantly, pose queries. The Diagrammatic Query System has a three-tier architecture [16]. The GUI system is the representation layer, and the CORAL deductive database acts as the DB layer. The translation system resides between the GUI system and CORAL deductive database. Hence, the GUI system is the upper level system, or the client, of the translation system. It fetches system schemas and user-defined views through the translation system. Internally, the translation system is divided into two parts. One is the translation engine, the other is the rest of the translation system. The translation engine utilizes the rest of the translation system to fulfill the tasks requested by the GUI system. So we see the translation engine as the actor of the rest of the translation system.

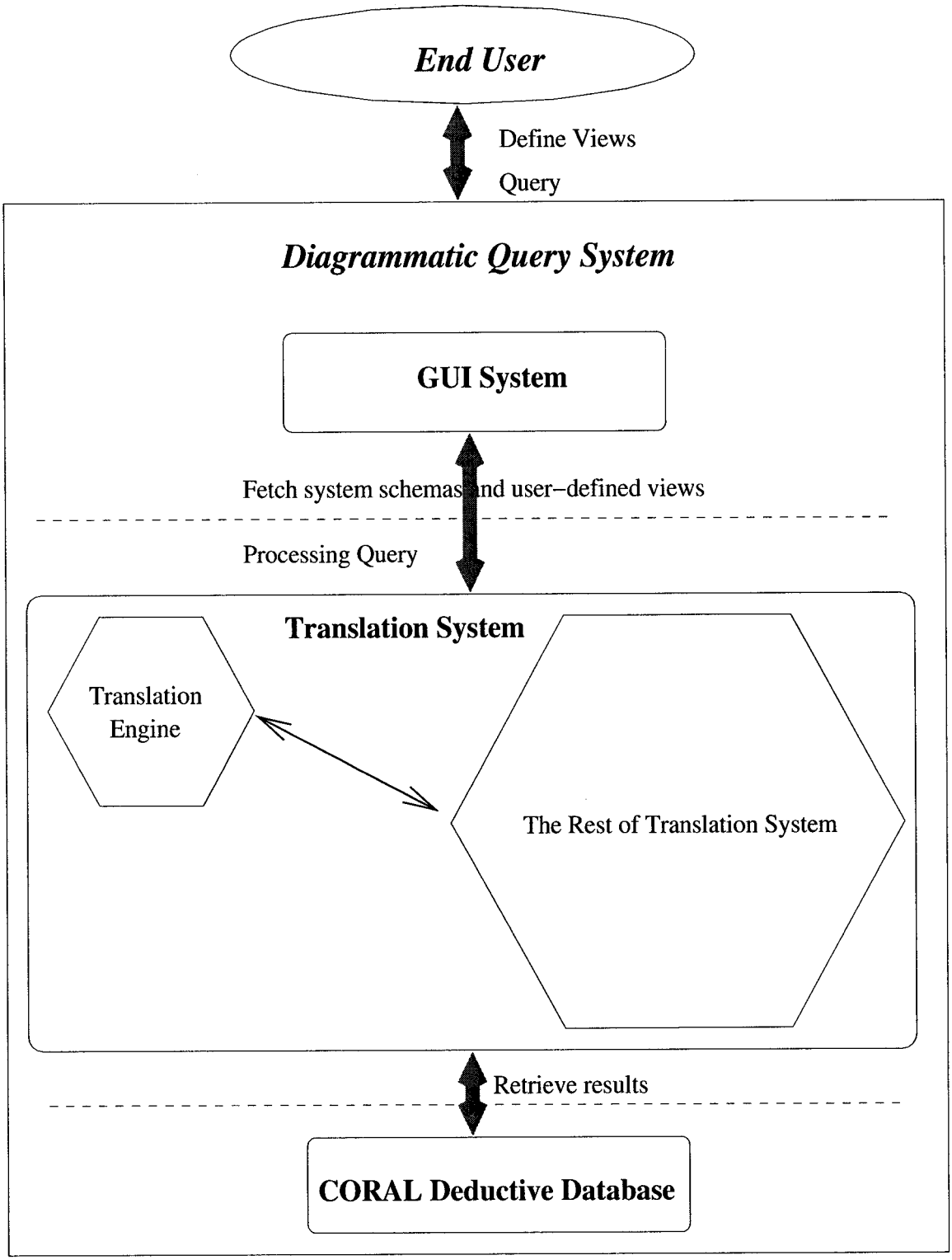


Figure 12: Context Diagram of The Translation System

4.2 Requirements and Incremental Plan

This section defines the complete requirements of the translation engine through a series of use cases. It describes their characteristics and functions, partitions the development, and derives the resulted increments.

4.2.1 Requirements Definition

4.2.1.1 Overall Description

First, the translation engine should be able to initialize the rest of the translation system, fetch up-to-date underlying system schemas and user-defined views. Then if the initialization is successful, the translation engine receives requests from the GUI system. The requests are derived from diagrams drawn by end users adhering to the definition of the TGL. Basically, although the combinations may be varied, the requests can be classified into two categories:

View Defining: A new entity with its attributes can be specified; an edge with particular features can be defined; a blob that depicts new relationships of container node and contained nodes can be regulated.

Query processing: The instance of an entity could be found; also the values of a particular attribute of the entity could be listed; still maybe just a variable in the query context is interested; the possible results of an entire edge are wanted; the combination of the container entity and contained entities need to be listed.

4.2.1.2 Use Cases Specification

The translation engine is the actor of the rest of the translation system. For each use case, its goal, inputs, and outputs are recorded. The following use cases, illustrated in Figure 13, will guide through the projects and later maintenance.The GUI system development team can also refer to them for design.

The *context* mentioned below consists of a set of non-distinguished entities, edges, and blobs that accompany the distinguished one or ones.

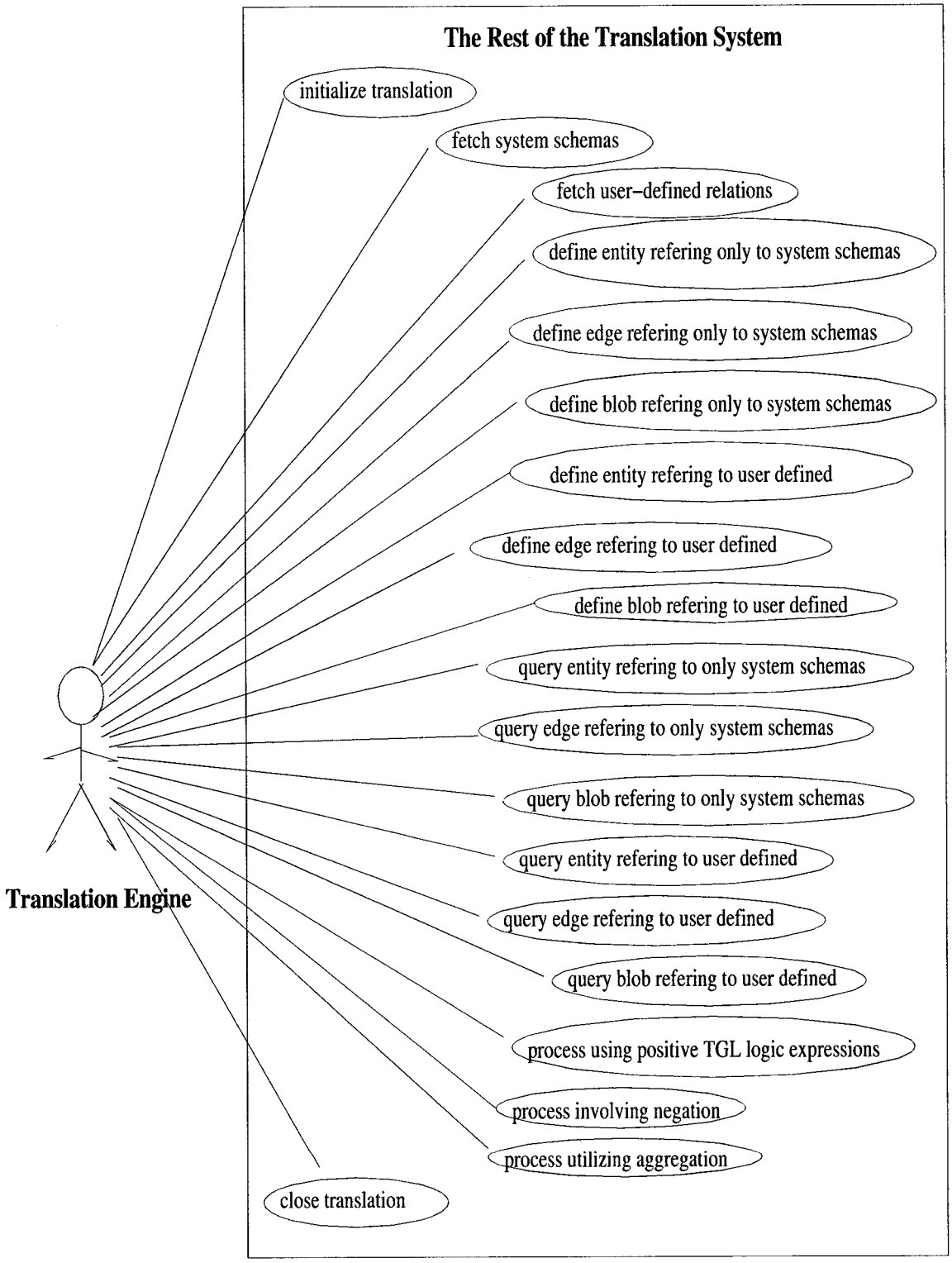


Figure 13: Use Case Diagram Inside The Translation System

Use case	initializeTranslation
Goal	prompts the translation system to be ready to communicate
Inputs	system time
Outputs	successful initialization
	not necessary since already initialized
	underlying systems not ready

Use case	fetchSystemSchemas
Goal	retrieves the schemes defined by system
Inputs	none
Outputs	an array of system schema's
	systems schema's do not exist

Use case	fetchUserDefinedRelations
Goal	retrieves the relations defined by users
Inputs	None
Outputs	an array of user-defined relations
	user-defined relations do not exist

Use case	closeTranslation
Goal	prompts the translation system to suspend
Inputs	none
Outputs	successful close
	translation system closed already.
	persistent information storing failure

Use case	defineEntityReferingOnlyToSystemSchemas
Goal	defines a new entity based on underlying system schemes
Inputs	distinguished node, context
Outputs	successful definition
	attempt to a overwrite system schema failure

Use case	defineEdgeReferingOnlyToSystemSchemas
Goal	specifies a new relationship of two entities based on underlying system schemes
Inputs	distinguished edge, context
Outputs	successful definition attempts to a overwrite system schema failure

Use case	defineBlobReferingToOnlySystemSchemas
Goal	describes a new relationship between an entity and a series characteristically similar entities based on underlying system schemes
Inputs	distinguished blob, context
Outputs	successful definition attempts to a overwrite system schema failure

Use case	defineEntityReferingToUserDefined
Goal	defines a new entity based on both underlying system schemes and user-defined relations
Inputs	distinguished node, a list of user-defined relation referred to, context
Outputs	successful definition attempts to a overwrite system schema failure

Use case	defineEdgeReferingToUserDefined
Goal	specifies a new relationship of two entities based on both underlying system schemes and user-defined relations
Inputs	distinguished edge, a list of user-defined relations used, context
Outputs	successful definition attempts to a overwrite system schema failure

Use case	defineBlobReferingToUserDefined
Goal	describes a new relationship between an entity and a series characteristically similar entities based on underlying system and user-defined schemes
Inputs	distinguished blob, a list of employed user-defined relation, context
Outputs	successful definition attempts to a overwrite system schema failure

Use case	queryEntityReferingToOnlySystemSchemas
Goal	looks for values of a or many existed entities based on underlying system schemes
Inputs	distinguished node or nodes, unit id, context
Outputs	results returned query pattern definition error

Use case	queryEdgeReferingToOnlySystemSchemas
Goal	searches keys for the from-node and to-node pair of a directed edge based on underlying system schemes
Inputs	distinguished edge or edges, unit id, context
Outputs	results returned query pattern definition error

Use case	queryBlobReferingToOnlySystemSchemas
Goal	finds the possible combination of a container node and a set of contained nodes that between them the blob relation or relations are satisfied based on underlying system schemes
Inputs	distinguished blob or blobs, unit id, context
Outputs	results returned query pattern definition error

Use case	queryEntityReferingToUserDefined
Goal	looks for values of a or many existed entities based on both underlying system schemes and user-defined relations
Inputs	distinguished node(s), unit id, employed user-defined relation list, context
Outputs	results returned query pattern definition error

Use case	queryEdgeReferingToUserDefined
Goal	searches value pairs for the from-node and to-node of interested edges with support of both system schemes and user-defined relations
Inputs	distinguished edge(s), unit id, utilized user-defined relation list, context
Outputs	results returned query pattern definition error

Use case	queryBlobReferingToUserDefined
Goal	finds the possible combination of a container node and a set of contained nodes that between them the blob relation(s) is satisfied based on both underlying system schemes and user-defined schemes
Inputs	distinguished blob(s), unit id, a list of employed user-defined relations, context
Outputs	results returned query pattern definition error

Use case	processUsingPositiveTGLLogicExpressions
Goal	defines and queries utilizing positive TGL logical expressions so that the query pattern is more powerful and expressive
Inputs	distinguished elements and context with TGL logical expressions, required information according to categories above.
Outputs	response or results returned TGL logical expressions definition error

Use case	processInvolvingNegation
Goal	defines and queries with negation involved
Inputs	distinguished elements and context with TGL logical expressions having negation, required information according to categories above.
Outputs	response or results returned TGL logical expressions definition error

Use case	processUtilizingAggregation
Goal	defines and queries with the support of aggregation functions
Inputs	distinguished elements and context described also through aggregation functions, required information according to categories above.
Outputs	response or results returned aggregation function misused

4.2.2 Incremental Plan

Through the incremental process, the translation system is partitioned into three manageable products. The complexity of these three products is increased one by one. Also the former always acts as a base for the later ones. The benefits achieved [22] are:

- Every increment is processed as a robust process and delivers a complete product. The functionality realized is part of the final product, so the early increment can help to understand what could be expected from the final product.
- Under the verification and support of the processing increment, the succeeding increment can process the more complex ones.
- A clear time schedule can be foreseen and evaluated along each pass of increment development.

Figure 14 illustrates the relationship among the use cases and assists in division of increments. The incremental plan constitutes three increments. The details of these increments are listed below. The division reflects the rule that processing goes from the simple to the complex, from the basic to the fulfilled. With each pass of increment, the rationale behind the participation is also included.

Increment One (Rationale — Deal with the simple use cases first, so basic functionality of the translation system can be analyzed and realized)

initializeTranslation (system time) {successful initialization, underlying system not ready}

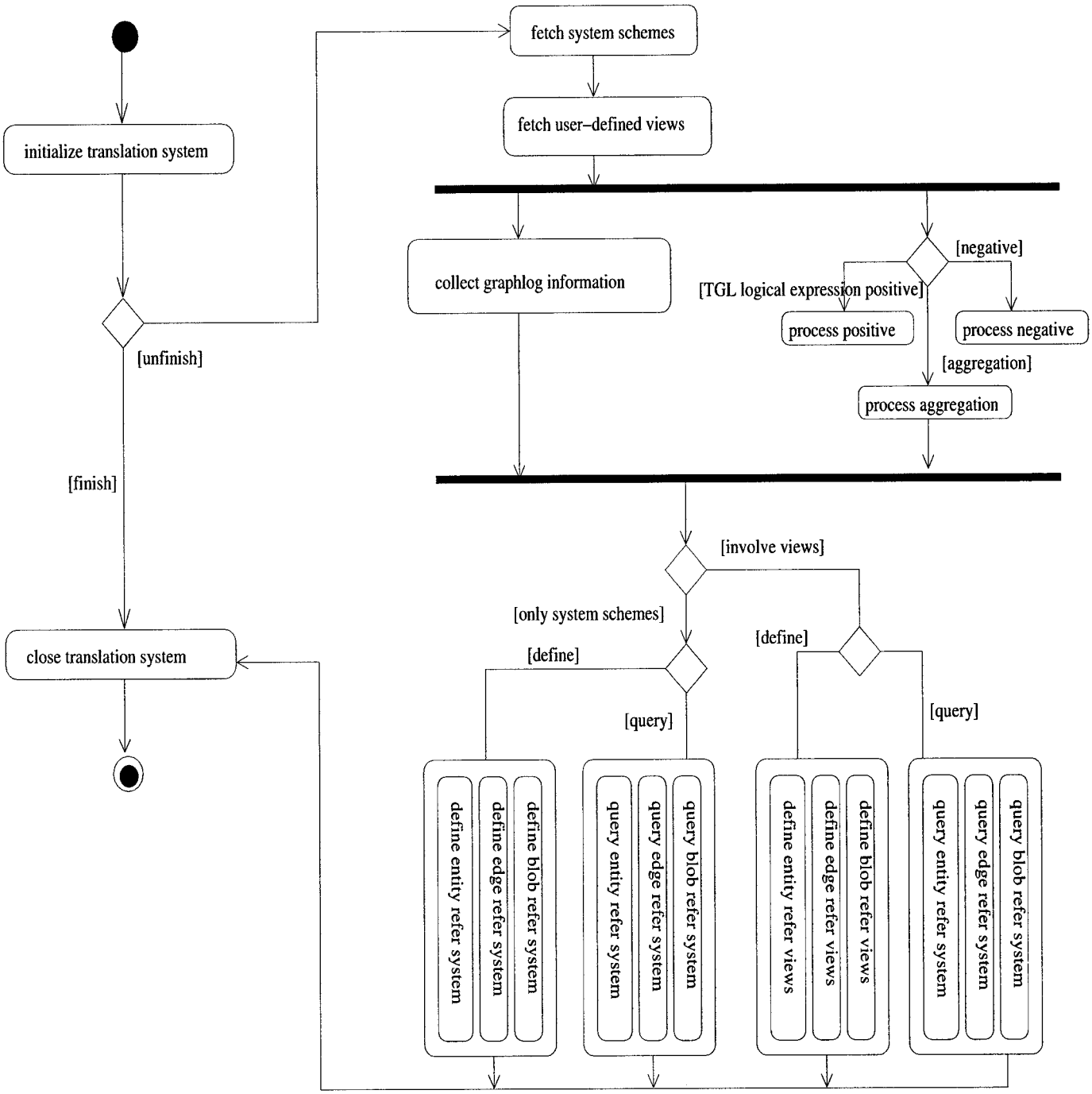


Figure 14: Activity Diagram of The Translation System

fetchSystemSchemas() {an array of system schemes, systems schemes do not exist}

fetchUserDefinedRelations () {an array of user-defined relations, user-defined relations do not exist}

defineEntityReferingOnlyToSystemSchemas (distinguished node, context) {Successful definition, Attempts to a overwrite system schema failure}

defineEdgeReferingOnlyToSystemSchemas (distinguished edge, context) {Successful definition, Attempts to a overwrite system schema failure}

defineBlobReferingToOnlySystemSchemas (distinguished blob, context) {Successful definition, Attempt to a overwrite system schema failure}

defineEntityReferingToUserDefined (distinguished node, a list of user-defined relation referred to, context) {Successful definition, Attempt to a overwrite system schema failure}

defineEdgeReferingToUserDefined (distinguished edge, a list of user-defined relations used, context) {Successful definition, Attempt to a overwrite system schema failure}

defineBlobReferingToUserDefined(distinguished blob, a list of employed user-defined relation, context) {Successful definition,Attempt to a overwrite system schema failure}

Increment Two (Rationale — processing queries is core function of the translation system. Users make a series of preparations to ensure the correctness and accuracy of the query pattern. In the second incremental pass, query function is realized, based on the groundwork accomplished in Increment One. Then after this stage, the translation system will possess sufficient parts to demonstrate its capability and be evaluated.)

queryEntityReferingToOnlySystemSchemas (distinguished node or nodes, unit id, context) {results returned, query pattern definition error}

queryEdgeReferingToOnlySystemSchemas (distinguished edge or edges, unit id, context) {results returned, query pattern definition error}

queryBlobReferingToOnlySystemSchemas (distinguished blob or blobs, unit id, context) {results returned,query pattern definition error}

queryEntityReferingToUserDefined (distinguished node(s), unit id, employed user-defined relation list, context) {results returned, query pattern definition error}

queryEdgeReferingToUserDefined (distinguished edge(s), unit id, utilized user-defined relation list, context) {query0501, results returned, query pattern definition error}

queryBlobReferingToUserDefined (distinguished blob(s), unit id, a list of employed user-defined relations, context) {results returned, query pattern definition error}

Increment Three (Rationale — process the most complex use cases of the translation system. TGL logical expressions provide more power and more convenient way to declare the query pattern, but additional processes are needed to incorporate the expressions into the program. Further, negation and aggregation are two error-prone fields in logic programming, extra efforts have to be made to ensure correctness.)

processUsingPositiveTGLLogicExpressions (distinguished elements and context with TGL logical expressions, required information according to categories above) {response or results returned,TGL logical expressions definition error}

processInvolvingNegation (distinguished elements and context with TGL logical expressions having negation, required information according to categories above) {response or results returned, TGL logical expressions definition error}

processUtilizingAggregation (distinguished elements and context described also through aggregation functions, required information according to categories above) {response or results returned, aggregation function misused}

Along the development of increments, the use cases presented in the generic requirements will be augmented with detail scenarios and corresponding pre-conditions

and post-conditions. Together a conceptual structure for the entire translation system will be developed.

4.3 Increment One Process

This section presents the course of analysis, design, implementation, and test for the indispensable functions of the translation system. These functions serve to connect the translation system, retrieve persistence schemes, and declare new relations.

4.3.1 Increment One — Analysis

Whereas requirements show the exterior consequences the user wants to acquire through the interactions with the system, analysis reveals the domain logic beneath the problems and reflects the development view compared with the user view of the requirement.

At this stage, use cases are reconsidered and refined from the technical point of view. However, goals, inputs, and outputs will not be listed again, and these can be referred to the Requirement and Incremental Plan section. In fact, the goals are detailed with scenarios.

4.3.1.1 Use Cases refinement

This part refines use cases for Increment One.

Use case	initializeTranslation
Scenario 1	The GUI system informs the translation system to be ready. After received the instructions, the translation engine warms up the translation system, and asks for the files that store persistent system schemes from the lower layer system.
Pre-condition	The translation system is idle and empty.
Post-condition	The translation system is prepared and system schemes files updated.
Exception	The lower layer can not be called, required files not updated.
Scenario 2	The remote upper layer system wakes up the translation system. The translation system is initialized. Connection between the host and client are established. Also the host obtains system schemes files from lower layer system.
Pre-condition	The translation system is sleeping.
Post-condition	Connection between client and server are established, system schemes info are retrieved from lower layer.
Exception	Connection can not be initialized. No system schemes recreated.
Use case	fetchSystemSchemas
Scenario	The upper layer system acquires run-time system schemes structure from the translation system. The translation engine reorganizes the system schemas files into the structure agreed by the upper layer system, and return it.
Pre-condition	The translation system is ready and system schemes files updated.
Post-condition	The translation system keep the system schemes and the upper layer have the system schemes information.
Exception	There is no any existing system schemes.

Use case	fetchUserDefinedRelations
Scenario	The upper layer system acquires predefined relations of users. The translation system retrieves existing user-defined relations information and return to upper system.
Pre-condition	The translation system is ready and has knowledge of where to retrieve user-defined relations.
Post-condition	The upper layer have the user-defined relations.
Exception	There is no any existed system schemes.
Use case	closeTranslation
Scenario	The local upper layer system tells the translation system to turn off. After received the instructions, the translation system record run-time structure into disk.
Pre-condition	The translation system is awaiting for new task.
Post-condition	Persitant data is restored.The translation system is idle now.
Exception	The physical storage fails.
Use case	defineEntityReferingOnlyToSystemSchemas
Scenario	The upper layer system recognizes the entity distinguished by user and collects assistant elements expressed based on system schemes. Then the upper layer system transforms these information into TGL program. Next, the upper layer system drives the translation system with the program. The translation system parsed the program, maps it to a CORAL program named by the relation name, and return a message indicating the successful creation.
Pre-condition	A TGL program including the information of the emphasized entity(name, attributes) and elements that form the context.
Post-condition	A new CORAL program that defines the entity is created.
Exception	The new entity name is the same with some system schema, error is returned.

Use case	defineEdgeReferingOnlyToSystemSchemas
Scenario	The upper layer system has the label, the from-node and to-node of the distinguished edge as the distinguish part of the TGL program. Then it records the aiding information as the content part. Next it calls the translation system to translate the TGL program into a CORAL program. After defining the CORAL program, the translation system tells back the success.
Pre-condition	A TGL program including the information of the emphasized edge(label, from-node, and to-node) and elements involved in the context.
Post-condition	The edge definition program is stored through the system.
Exception	The edge label repeats a system schema name, error is sent back.
Use case	defineBlobReferingToOnlySystemSchemas
Scenario	The upper layer system states the highlighted blob with its label, container node, and a set of contained nodes in the TGL program. Then encloses other context element information into the program. At the service of the upper layer system, the translation system sets up a corresponding CORAL program and sends back a successful response.
Pre-condition	A TGL program including the information of the emphasized blob(label, container node, and contained nodes) and elements involved in the context.
Post-condition	The blob definition program is stored through the system.
Exception	The blob label reuses a system schema name, error is sent back.

Use case	defineEntityReferingToUserDefined
Scenario	The upper layer system recognizes the entity distinguished by user and records the circumstances. The query pattern employs both system schemes and user-defined relations. Next, the upper layer system calls the translation system with the TGL program that describes the information. The translation system parsed the program, maps it to a CORAL program named by the relation name, and records the new relation name and related user-defined predicates into run-time structure. A message is returned to indicate the success.
Pre-condition	A TGL program including the information of the emphasized entity(name, attributes) and elements that form the context.
Post-condition	A new CORAL program that defines the entity is created, the name and linked user-defined relations recorded.
Exception	The new entity name is the same with some system schema, error is returned.
Use case	defineEdgeReferingToUserDefined
Scenario	The upper layer system transfer A TGL program and related user-defined relations to the translation system. The TGL program has the information of the distinguished edge and other aiding elements in the circumstance. The translation system translates the TGL program. Also it records the relationship of the new defined edge and used non-system schemes.
Pre-condition	A TGL program including the information of the emphasized edge(label, from-node, and to-node), elements involved in the context, and non-system schemes used.
Post-condition	The edge definition program is stored through the system, the link between the new user-defined and existing ones is dynamically kept.
Exception	The edge label repeats a system schema name, error is sent back.

Use case	defineBlobReferingToUserDefined
Scenario	The upper layer system states the highlighted blob with its label, container node, and a set of contained nodes in the TGL program. Then encloses other context element information into the program. In addition, it indicates which user-defined predicates are used. At the service of the upper layer system, the translation system sets up a corresponding CORAL program, records the related user-defined predicates, and sends back a successful response.
Pre-condition	A TGL program including the information of the emphasized blob(label, container node, and contained nodes) and elements involved in the context.
Post-condition	The blob definition program is stored through the system.
Exception	The blob label reuses a system schema name, error is sent back.

4.3.1.2 Analysis Class Diagram

The class diagram shown in Figure 15 is based on the use cases in Increment One and thus the model is confined to the classes of the current increment. Note that only the top level classes are in this diagram to illustrate the process outline, and sublevel classes and their functions are depicted in the following sections. Note that the naming convention for the classes in engine package are prefixed with the character 'X', this is used to differentiate from classes with similar names in other packages.

4.3.2 Increment One — Design

4.3.2.1 Conceptual Structure of Increment One

The reasonable motivations to formulate a conceptual structure of the translation system are: ensuring the development to align with domain logic, guiding decisions of the rest process, and acting as communication vehicles with other layers.

At this stage a structure for use cases in Increment One is established and shown in Figure 16. Along with succeeding increments, the structure will be augmented.

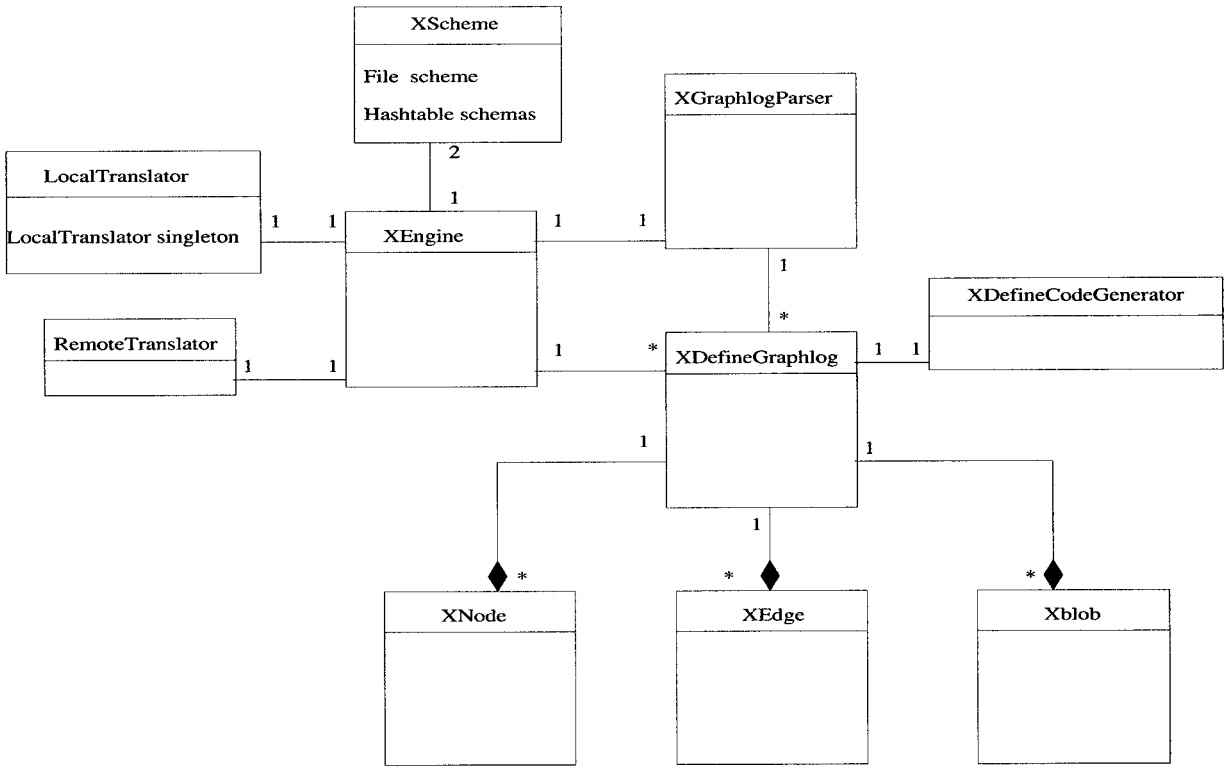


Figure 15: The Analysis Class Diagram of Increment one

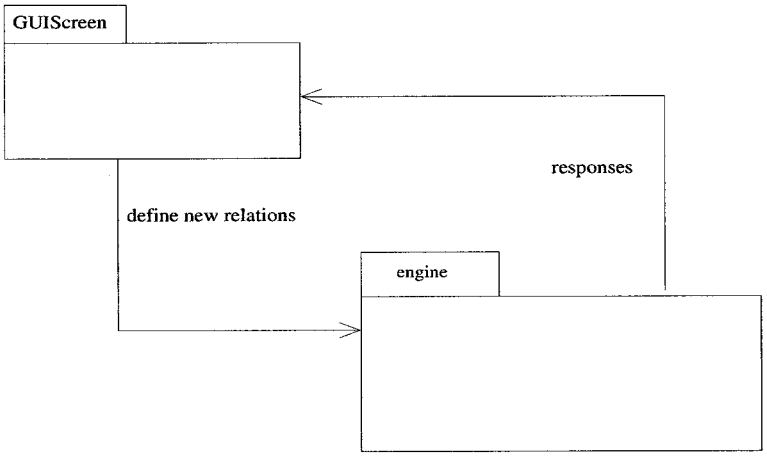


Figure 16: The Conceptual Structure at Stage of Increment One

4.3.2.2 Translator Templating Methods

TRANSLATOR is an abstract base class for LOCALTRANSLATOR and REMOTETRANSLATOR. It is responsible for initializing, prompting the underlying system to provide up-to-date shared information, fetching translation persistent states, transferring back response and translation results. In fact, many details for the above transactions are significantly different for LOCALTRANSLATOR and REMOTETRANSLATOR. One solution is leaving all the methods of TRANSLATOR abstract until the subclasses give the concrete algorithms. However, if we pay attention to the fact that the skeletons of these algorithms in both subclasses are similar, a better alternative could be derived. By using the *Template Method design pattern* [8], TRANSLATOR could define the structures of the methods, and subclasses only need to flesh out the distinct parts. The advantages over the first solution are

- The subclasses need not hard code all the details separately.
- The commonalities are abstracted and made reusable.
- A clearer structure is determined at design time.

4.3.2.3 LOCALTRANSLATOR Being Singleton

Translators will transform user-defined relation from the TGL program format to executable CORAL programs. It makes sense to keep the corresponding CORAL program. First, the user will probably check the correctness of the definition or utilizing it immediately after the relation is defined. Keeping the CORAL program will save another run. Second, a user-defined relation may be used many times, keeping the translated CORAL program will save time and increase performance.

The next decision made is to determine the place and structure to store these user-defined CORAL programs. One choice is to reserve disk spaces for every user, consequently, users do not share their defined relations with others. The only common support comes from system schemes. Another choice is to reserve a disk space for all users so that each user can share other user's definition. The translation system takes on the later choice. The reasons backing this decision are that:

- Many users may define similar relations. The first choice may cause many copies of a single user-defined relation. This is not efficient. Moreover, it precludes

reusabilities.

- Some users are just one-time visitors or seldom use the system, it it wasteful to preserve space for them.
- The first choice makes file system management complex, while managing files is not the main responsibility of the translation system.

However, the second choice is not a golden solution, either. There exists the problem that a user may replace others' definition by mistake. We solve this problem by providing a list of all existing user-defined relations to users before they define their own. Not only could it let users reuse resources, but it prevents their carelessly overlapping others' artifacts.

Another possible problem caused by the later choice is that more than two executions of the translation system may overlap. When many relations are employed and the TGL program gets complex, one execution can last a period of time. With modern computer systems, the overlapping of two executions is quite possible. Multiprocessing is a popular technique of contemporary operating systems, and CPU utilization is maximized by arranging a process on each CPU. Even if there is only one CPU, its time may probably be divided into intervals that let each process have a chance to run. As a consequence, two definitions of a relation may be executed interchangeably and the final CORAL program is a strange mixture.

The solution of the problem is to apply the *Singleton Design Pattern* [8] to the LocalTranslator. Through letting the LocalTranslator class itself to keep track of its sole instantiation, it is guaranteed that there exists only one instance. Also the method is public and can be accessed from any global point. Notice that the constructor of the LocalTranslator is protected instead of public. Thus clients outside the package can not instantiate the LocalTranslator by using its constructor.Also the protected accessibility ensures the extensibility of the Translator class and thus clients are able to use an extended instance without changing the code.

4.3.2.4 XEngine As Facade of Translation Details

The translation process is complex and the client is not interested in how to exchange messages at low level, either. A high level interface that encapsulates the details is

preferred, So XENGINE class takes the role.

The translation process consists of scanning files into tokens, analyzing the grammar of the token sequence, producing internal data structures, generating target code under the aid of the internal data structure, executing the generated code on the target system, fetching the result, parsing the result, transforming the result to acceptable format, and returning the result. Each step needs a set of classes to support. Although it leads to much more flexibility and enhances reusability through providing the interface to each group of classes and permitting customization, here the client cares only for the result with the acceptable format, but not the configuration details. Consequently, leaving a wealth of interfaces to be configured only causes complexity to the client.

Therefore, a facade is introduced through class XENGINE. The design conforms to *Facade Design Pattern* [8]. The client need only to pass the TGL program file for a graphlog to an XENGINE object, then it can receive the names of generated CORAL programs. The client needs even not think about how the file names are abstracted from the CORAL program files. It is the XEngine object that takes care all the translation process and supplementary activities. The client sends message through the *returnResults* method of XENGINE class. The method encapsulates all the translating transactions.

By using the facade pattern, the translation system is able to decouple the client from translation details and promote the independence of the translation system.

4.3.2.5 Runtime Data Structure and *Flyweight Design Pattern*

A graphlog consists of a set of query patterns(defineGraphlogs and/or showGraphlogs), while a query pattern constitutes nodes, edges and blobs.There are still intricate relationships between the nodes, edges, and blobs: a node may be the end of several edges and some blobs at the same time; an edge has two nodes as the ends; a blob has a great number of contained nodes.Furthermore, an element may modify the attributes of other elements. For instance, an edge with label "is_a" emphasizes that the from-node and end node should have the same key. All these features indicate that the data structures representing the elements should correctly reflect the consitution and be easily referred by one another.

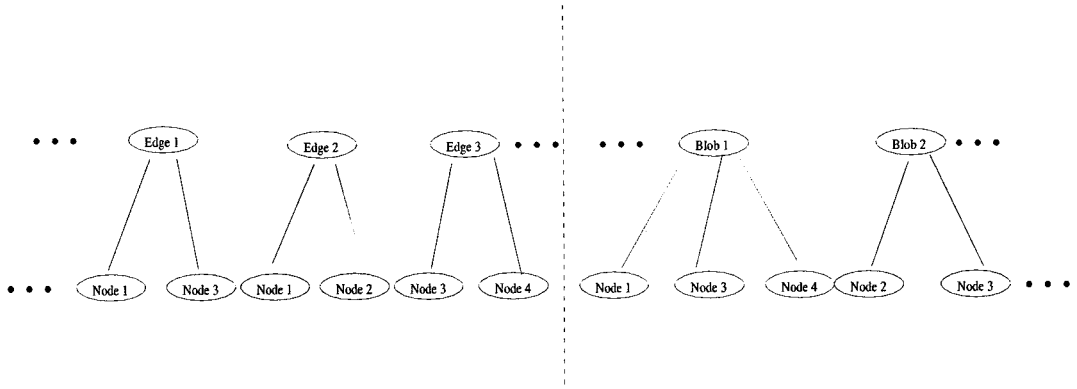


Figure 17: Organization of Nodes, Edges, and Blobs Before Applying Flyweight Pattern

In addition, the number of elements could be huge, which requires efficient construction and access.

Flyweight Design Pattern [8] grants an effective method to solve similar problems. The basic idea is: first, to create all lowest level elements and manage them in a pool. Then, let elements of higher levels be constructed with reference to the basic. Last, to decorate the the basic and reuse them at runtime.

In this application, nodes are the lowest level of the construction, whereas edges and blobs have nodes as construction elements. So for a particular graphlog pattern, all nodes are built independently and collected together. Whereas edges and blobs are built with their specific information and references to-nodes in the collection. Thereby, all the nodes are shared and copies of nodes are saved for edges and blobs.

The situations before applying *Flyweight Design Pattern* and after applying the pattern are illustrated in Figure 17 and Figure 18 respectively. It is clear that the number of node objects after sharing nodes among edges and blobs is far less than that of before sharing. Plus,

The benefits achieved through the design are:

- Memory space for storing objects is soundly saved.
- Total number of instances are significantly reduced through sharing.
- The data structures of node, edge and blob are both elegant and efficient.
- The maintenance of node objects with same states is saved when one of them is updated.

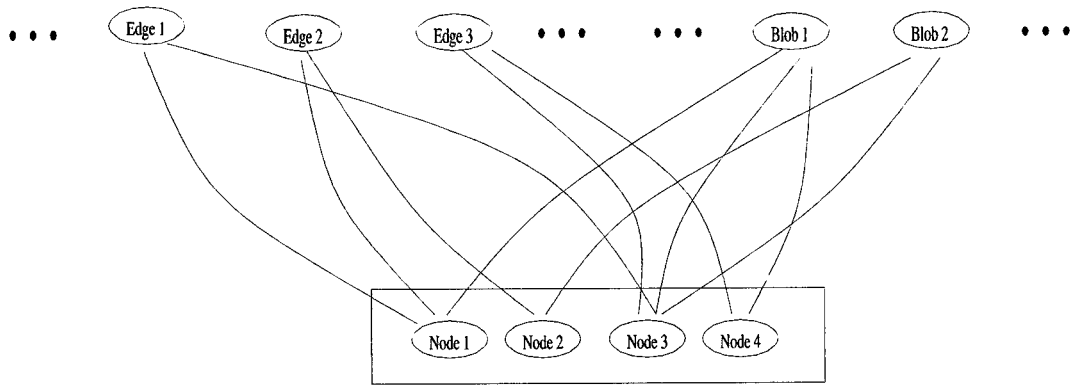


Figure 18: Organization of Nodes, Edges, and Blobs After Applying Flyweight Pattern

```
class EnclosingClass{
    ...
    class ANestedClass {
        ...
    }
}
```

Figure 19: Nested Class

4.3.2.6 Inner Class Richening functionality

The Java language has a powerful design and implementation vehicle: Nested Class [21]. A nested class is a class that acts as a member variable of another class. It is illustrated in Figure 19.

An inner class is a non-static nested class. It has unlimited access to all the members of its enclosing classes, including the static and the non-static. Consequently, the enclosing class increases its functionality with the aid of the inner classes. Nonetheless, an inner class can not define any static members of its own because it associates with an instance of its enclosing class. An inner class only makes sense in the context of its enclosing class.

In the translation system, XGRAPHLOGPATTERN objects collect their basic states along the procedure of parsing the corresponding TGL program. But these states are not the final states that are utilized when generating executable CORAL programs. Further computation and organization are necessary to form the final states based on the basic states. It is helpful and can increase performance if these repeatable computation and organization could access the basic states with least restrictions.

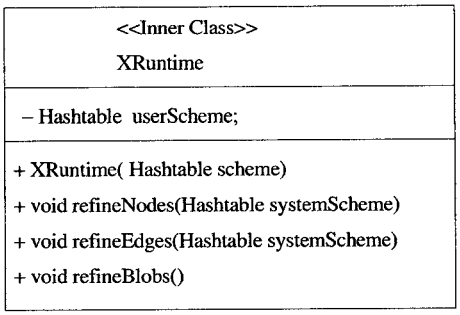


Figure 20: Inner Class XRuntime in Increment One

An inner class of XGRAPHLOGPATTERN meet the requirements metioned-above. First, it can access the node, edge and blob members freely and thus can assign up-to-date states to them after modification without complex message sending. Second, the abstraction of the computation is declared as an inner class's member methods and can be reused. Figure 20 depicts the design of the XRUNTIME class that plays the role of an inner class in the XGRAPHLOGPATTERN class.

Assisted by the inner class XRUNTIME, the XGRAPHLOGPATTERN could refine its member variables xNodes, xEdges, and xBlobs dynamically.

4.3.2.7 Tokenizing the TGL program

This section introduces the design of scanning the TGL program and breaking it down into tokens.

Token is the fundamental unit of a language. It represents a logical chunk of characters that has meaning in a particular language. Generally, it could be a number, a word, or a punctuation. As a human reader, the first step for a computer utility to analyze a text representation in a language is to find all possible tokens. In terms of compiler technology, the process of dissecting text is called lexical analysis, too. It is complex and impractical to identify the tokens while recognizing the grammar of the language. It is against the rule of Object-Oriented Design that an object only care about its own responsibilities.

Tokenizers' responsibilities A XTOKENIZER class will be responsible to locate all tokens in TGL programs. The role of the XTOKENIZER class is as follows:

- Identify numbers.

- construct words from letters, digits, and special characters allowed.
- Locate one character symbols like ' >', ' −' .
- Allow multicharacter symbols, such as " < /"*and*" − −"
- Recognize whitespaces as token delimiters.
- Shape double quoted string.
- Ignore comments beginning with " − −".

It reads roughly like a human being does, especially it finds precisely where a token begins and ends. For example, the tokenizer will view the string " < */graphlog* > " as containing 3 tokens: a multicharacter symbol, a word, and a symbol. That is,(< /), (*graphlog*), *and*(>).

An XTOKENIZER object does not hold these tailored logical strings. It forms XTOKEN objects using this information. So the XTOKEN class does not determine what is or what is not a token, but it does hold the results determined by the tokenizer.

Tokenizing following State Design Pattern [8] The XTOKENIZER class has a set of internal state objects that relates to various types of tokens. And an object of the TOKENIZER class alters its behavior when its current internal state object changes.

The abstract class that represents the states of recognizing tokens is XTOKEN-STATE. The interface common to all classes that represents different tokenizing states is *nextToken(PushbackReader r, int c, XTokenizer t)*. There are a total of five states for tokenizing TGL program: XBeginOrEndState, XValueState, XWhitespaceState, XCommentState and XErrorState. The corresponding class hierachy is illustrated in Figure 21. The concrete states implement their state-specific behavior respectively.

A state is associated with a set of characters. The states and their linked characters are listed in Table 7.

To decide what the current state is, the XTOKENIZER class looks up in the table with the character it just read. After a state is matched, the TOKENIZER object changes its current state by sending message to an appropriate XTOKENSTATE object. Besides the character read by the TOKENIZER object, the TOKENIZER object

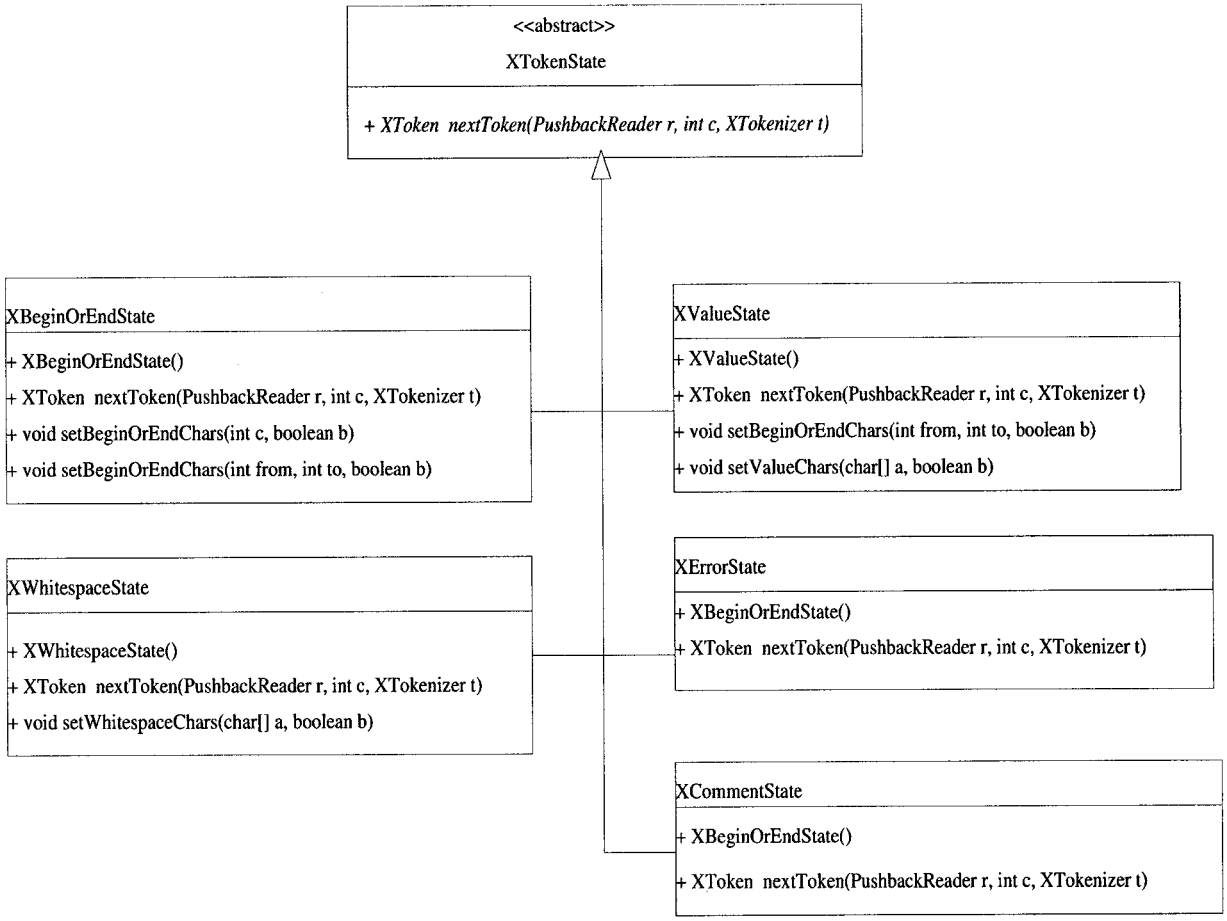


Figure 21: Tokenizer States Class Hierachy

From	To	State
'<'	'<'	BeginOrEndState
'"'	'"'	ValueState
'&'	'&'	ValueState
'-'	'-'	ValueState
'='	'='	ValueState
'('	'('	ValueState
'a'	'z'	ValueState
'A'	'Z'	ValueState
'0'	'9'	ValueState
'-'	'-'	CommentState
The rest characters	The rest characters	ErrorState

Table 7: Tokenizer States and Associated Characters

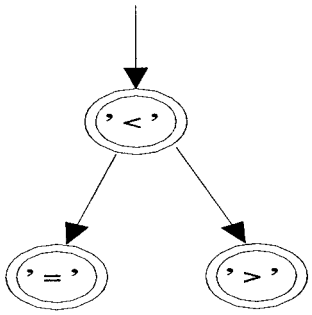


Figure 22: Potential Paths of recognizing a Symbol

itself is passed to the state. In addition, all the tokenizer states share one single PushbackReader by accepting it one by one.

The PUSHBACKREADER is usually generated in the XTOKENIZER constructor, still a flexibility is provided through *setReader()*.Thus the TOKENIZER object can be reused by simply changing the resource. PUSHBACKREADER has a special ability that it can put back a character to the resource after reading it. This is useful in the situation when the next character must be read to judge whether the end of a token is reached, but unfortunately, the current other than the next character is the end, so the next character must be sent back to the resource for another token. Recognizing a symbol is a good example. All the three symbols are legal in TGL: " < ", " <= ", *and*" <> " and Figure 22 shows the potential paths of recognizing a symbol.

When an XVALUESTATE object encounter the character ' <', it looks to see whether the next character in the input stream matches ' =' *or*' >' . If there is a match, the new symbol token is fixed. Otherwise, the symbol is just a ' <' and whatever the character looked over is, it has to be put back into the input stream. The PUSHBACKREADER class helps this by its method *unread(int c)*.

4.3.2.8 The XToken Class

The XTOKEN class serves as a container for a string of text chunked by an XTOKENIZER object.

An XTOKEN object has a type and a value. The XTOKEN class takes the following token types as member variables: XTT_EOF, XTT_BEGIN, XTT_END, XTT_VALUE, XTT_COMMENT, and XTT_ERROR.

These types help XParsers to distinguish the tokens in terms of the roles they play

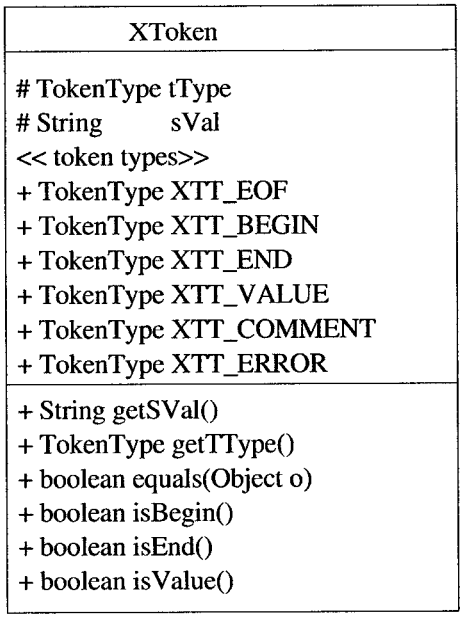


Figure 23: XToken Class Diagram

in the TGL program. So a tag like " $<$ *graphlog* $>$ " is a token of type XTT_BEGIN, and the value of the token is the string "graphlog". Moreover, an example of a token with type XTT_END could be " $</$*graphlog* $>$ ".

Although a token need not to identify itself, it does need tell itself from others. The *equal(Object o)* method addresses the function. Notice that the parameter is an instance of Object class, but not of *XToken* class. The reason is that by default XTOKEN class inherits from the class OBJECT, which is the ancestor of all the classes, and *equal(Object o)* is a protected method that need to be specified by any subclasses.

In addition, an XTOKEN object should be able to claim its own type. The *isXXX()* methods that return boolean values are designed for this purpose. Figure 23 shows the class diagram of XTOKEN.

Summarily, while scanning a TGL program, an XTOKENIZER object distinguishes tokens relying on its internal states. Then it forms XTOKEN objects that represent the recognized tokens. The overall design of the tokenizing process is illustrated in Figure 24 .

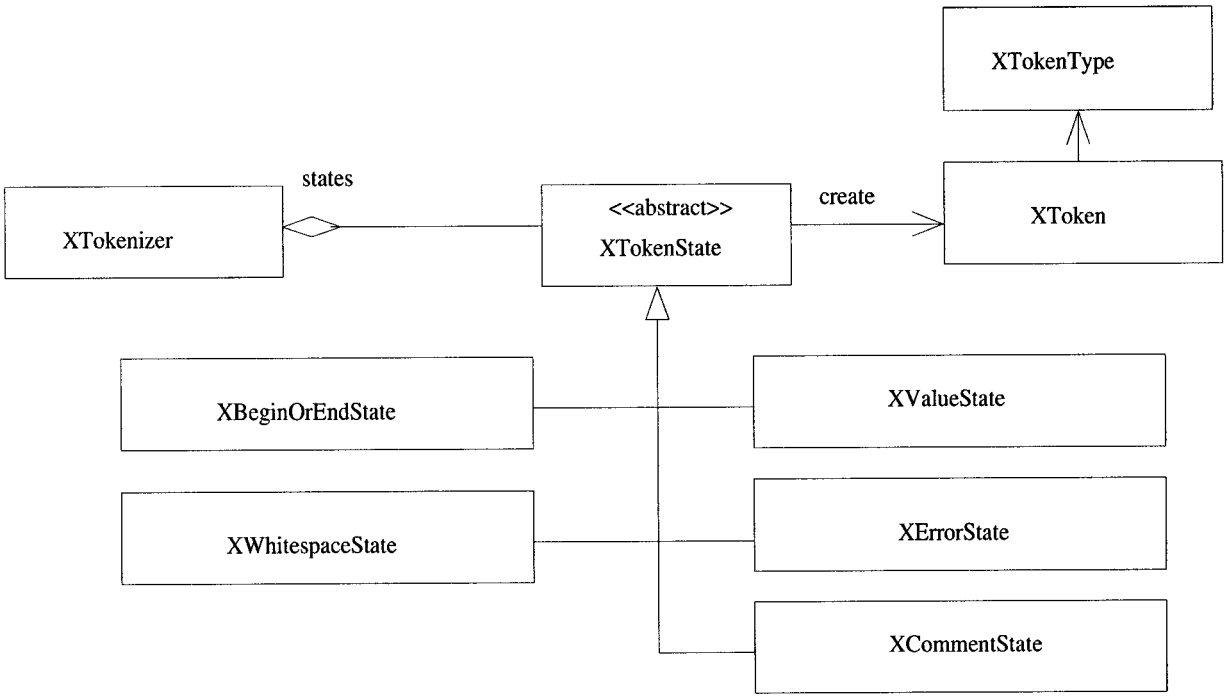


Figure 24: Tokenizing TGL Programs Class Diagram

4.3.2.9 Parsing hierachy for TGL program and *Composite Design Pattern* [8]

A set of query patterns are organized into a single TGL program; each query pattern has a distinguished part and a content. Furthermore, the distinguished part and the content are composed of nodes, edges, and blobs. Moreover, the number of possible constructions is large. The relation that a higher level component is made up of lower level components suggests a composition structure.

Consequently, primitive classes can be defined for parsing the fundamental components like nodes, edges, and blobs. Then container classes for parsing the distinguished part and content can be built from these fundamental parsers. In turn, these parsers can be grouped to form larger parsers, for defineGraphlog and showGraphlog.

However, containers have some different behaviors that composing items do not have, although the procedure of breaking a TGL program into tokens and reconstructing these tokens into runtime data structure for further process is the same.

To distinguish these objects will notably increase complexity and significantly reduce the reusability. *Composite Design Pattern* presents a reasonable solution to the problem: let the super class possess behaviors of both the simple components and

the complex components. Thereby, the developer need not specify classes to deal with a particular parsing process at design time, but leave it until run-time binding. Applying to our system, the super class is XPARSER class. And the leaf classes are XNODEPARSER, XEDGEPARSER, and XBLOBPARSER. Next, XCONTENTPARSER, XDISTINGUISHDEFINEPARSER, XDISTINGUISHSHOWPARSER are composed of XNODEPARSER, XEDGEPARSER, and XBLOBPARSER. The continuous level components include XSHOWPARSER, which consists of a XDISTINGUISHSHOWPARSER and a XCONTENTPARSER, and XDEFINEPARSER, which consists of a XDISTINGUISHDEFINEPARSER and a XCONTENTPARSER. Then the highest level composition class is the XGRAPHLOG made of a XDEFINEPARSER and a XSHOWPARSER. All leaf classes make use of the method *getIDKey* to return the assigned id of the corresponding elements. Moreover, except for the XGRAPHLOGPARSER, all the rest of the parser classes share a member method *parse(XTokenizer, XGraphlogPattern)*. And the XGRAPHLOGPARSER has its own parsing method *parse(XTokenizer)* . All these methods are abstracted upon the tree to the top super class, XPARSER. Figure 25 shows the parsing class hierachy.

4.3.2.10 Generating Coral Program

With the states of refined XNODES, XEDGES, and XBLOBS objects for a define-Graphlog, generating a corresponding CORAL program is essentially to restate the query pattern in terms of CORAL language.

To define a new predicate, a typical CORAL program encloses the definition of the predicate in a module. The mark of the beginning of the module is the keyword *module* plus the name of the module and a period. And the keyword *end_module* and a period marks the finishing of the module. The predicate can be reused by other predicate outside the module through exporting it. Also the usage of the predicate is specified by pointing out whether the attributes should be bound or free. The symbol period specifies the end of a CORAL statement.

An example is shown in Figure 26. The program defines the predicate *creditMT2*. It can be used outside the module since it is exported by the module. The only attribute is free while making queries. The predicate, course(Code, _, Credits), appears in the rule body to support the definition. The variables Code and Credits must be

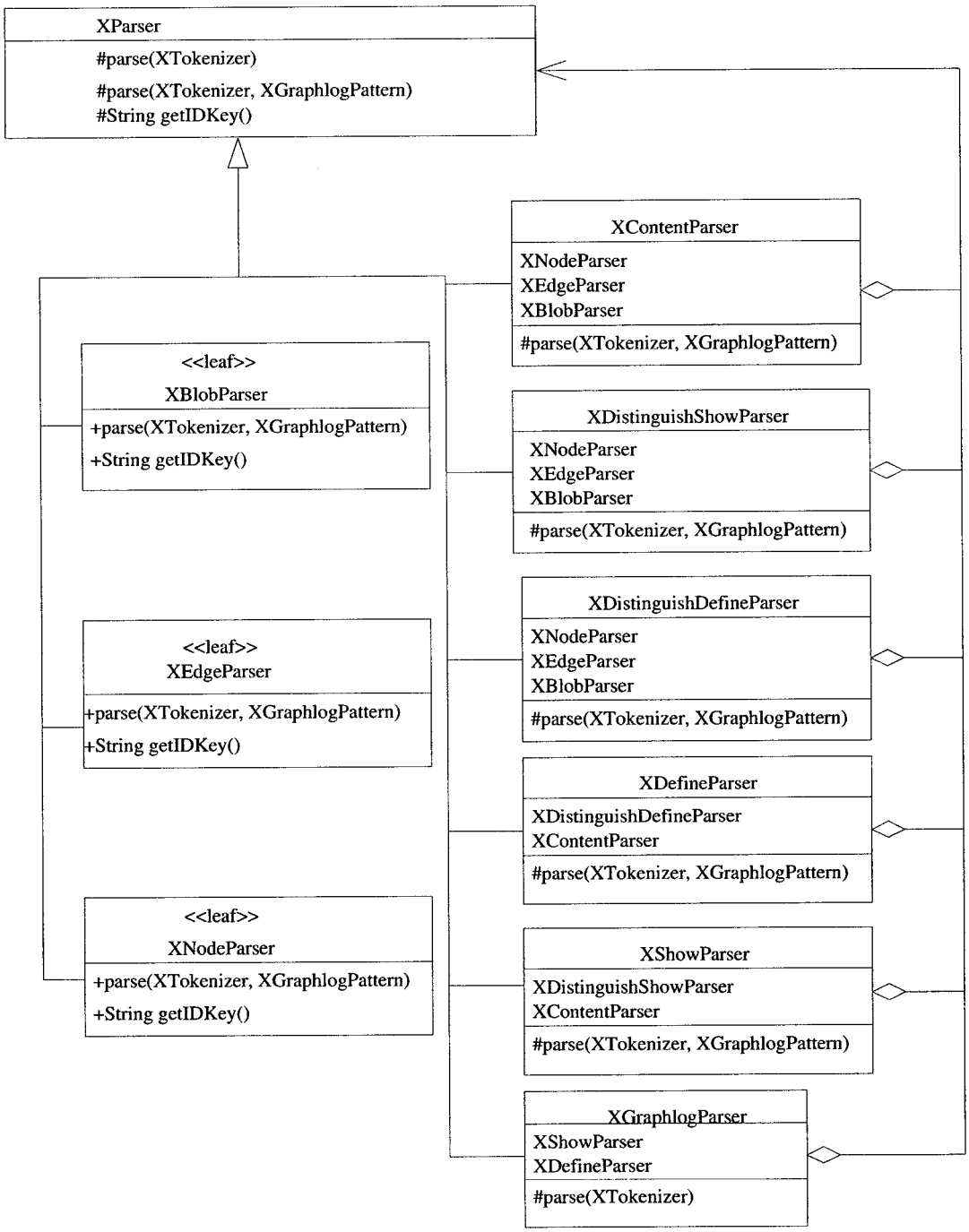


Figure 25: Class Hierarchy for Parsing TGL Program

```
module q0.
export creditMT2(f).

creditMT2(Code) :- course(Code,_, Credits),
  Credits > 2.

end_module.

?creditMT2(Code).
```

Figure 26: An Example Coral Program

bound before evaluating the rule. However, the second variable is an anonymous variable, and applies to any value at the position of facts during the evaluation. Notice that queries can be processed in a CORAL program as well.

So translating a defineGraphlog is transforming the TGL program into a CORAL module that exports the name of the distinguished elements and specifies the usage. XDEFINECODEGENERATOR class is responsible for the generation action. An instance of it is constructed with the refined XDEFINEGRAPHLOG object. Then the XDEFINECODEGENERATOR object abstracts useful messages and organizes the program through the method generateCode. By default, the names of the module and CORAL program are the same as the predicate. This facilitates the computation and search of a user-defined relation. XDEFINECODEGENERATOR class also provides the method *getModuleName* to pass the newly defined CORAL program name to the client for the purpose of updating records of user-defined relations.

4.3.2.11 Exception Handling

The exception hierarchy framework created by Java language considerably facilitates error-handling. Exceptions are categorized into checked exceptions and unchecked exceptions. The checked exceptions will be examined at compile time to ensure that they are dealt with by developers. And the unchecked exceptions are those occurring in run time and probably due to ill coding logic.

Translation System Exception Hierarchy A new tree of exceptions are defined by the translation system. These and other exception class libraries of Java language are wired into the system to react to the problems occurring at run time. The newly constructed exceptions follow the naming convention that the class names will end

Exceptions	Functions
TranslationException	the super class of all newly defined exceptions
AlreadyInitializedException	the translation system has been initialized and it is ready.
AlreadyClosedException	the translation system has been closed.

Table 8: Newly Declared Exception Classes Serving Increment One

with string "Exception".

If the translation process is interrupted at some point, corresponding exceptions will be thrown. By catching the exceptions, the client will understand what cause the interruption of the execution. There are set of reasons that may lead to exceptions:

- The underlying system that supports the translation system is not available.
- The file system may have I/O problems.
- The client set up unreasonable calls.

The TranslationException class is designed as the superclass of all newly defined exceptions for the translation system.

Newly Defined Exceptions in Increment One In Increment One, the newly declared exceptions and their functions are listed in Table 8.

4.3.2.12 Summary

The design delivery of Increment One is illustrated in Figure 27.

4.3.3 Increment One — Implementation

Implementation emphasizes the algorithms that realized the design. Put in another way, implementation of a system is responsible for fleshing out the member methods of classes.

4.3.3.1 Implementing Template Methods

Class TRANSLATOR has two template methods: *initialize* and *close*. They employ two protected methods *activateUnderlying* and *garbageCollect*, whose implementation

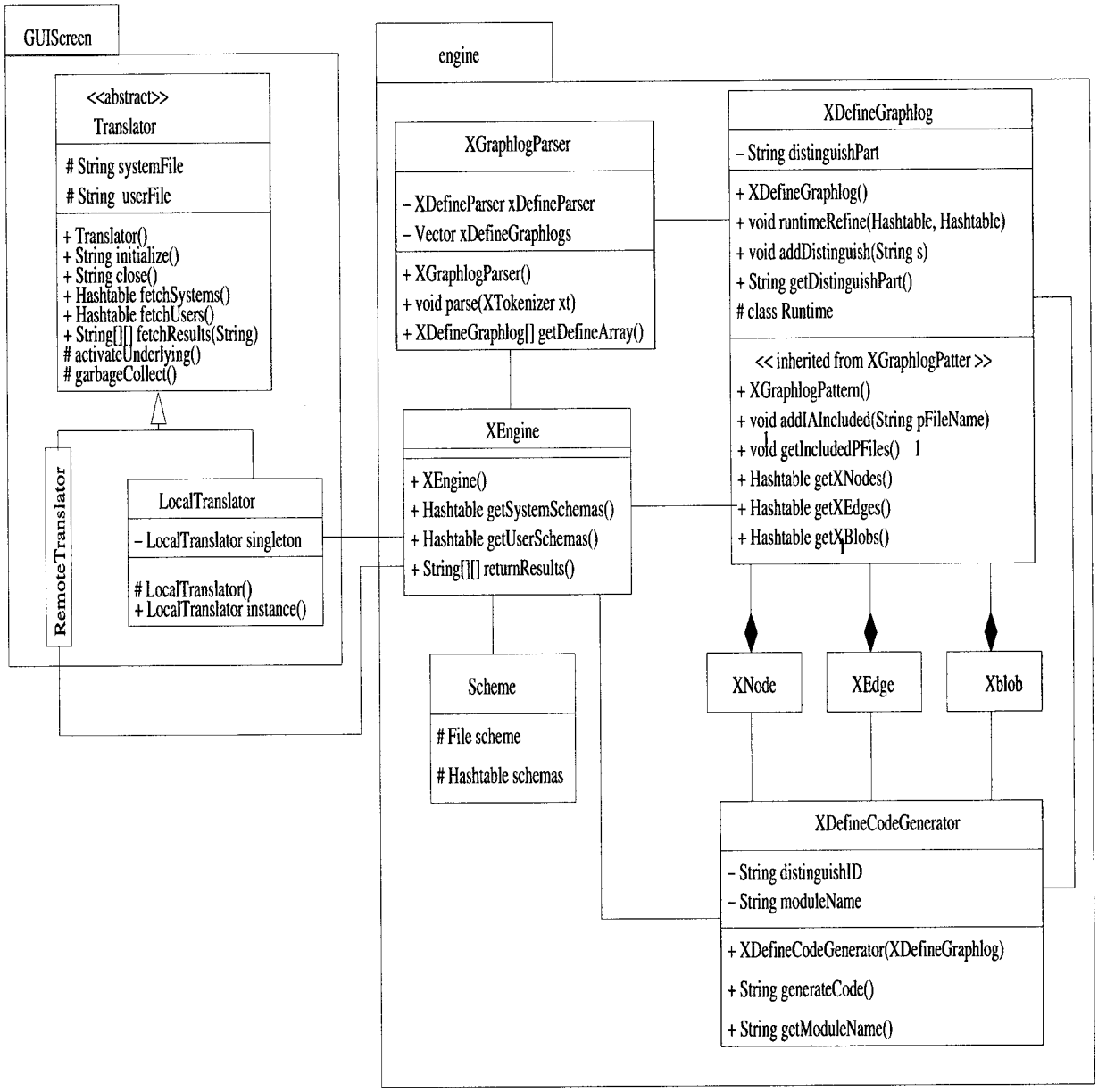


Figure 27: The design class diagram of Increment one

```
private static LocalTranslator singleton = null;

public LocalTranslator instance(){

 if(singleton == null){
  singleton = new LocalTranslator();
 }

  return singleton;
}
```

Figure 28: The Implementation of Class LocalTranslator Being Singleton

will be determined in the subclasses, respectively. Thus the subclasses keep template through overridden the *activateUnderlying* and *garbageCollect* methods. LOCALTRANSLATOR has a XENGINE object as a member variable. As the heart of the translation system, an XENGINE instance will keep many runtime data structures, which consume extensive resources. So the key point of inactivating the translation system is to release the resources from the XENGINE instance. We could recommend the garbage collector of Java Virtual Machine to collect it through setting the instance to null. Nonetheless, in this situation, it is not so simple because the XENGINE object in fact remains noncollectable if there are still references to it. So it must be ensured before setting the XENGINE object to null that all the references to the XENGINE object should be null. The task is fulfilled by *engine.garbageCollect()*.

4.3.3.2 Implementing Singleton

To ensure there are no system runtime conflicts, LOCALTRANSLATOR class is determined to provide only a single instance.

Clients will obtain the reference to the singleton exclusively through the member method *instance* of LOCALTRANSLATOR. The static member variable singleton can only be accessed by the LOCALTRANSLATOR class itself, and it is initialized to null by default. The method *instance* uses lazy initialization. That is, the instance singleton will not be created and kept at runtime until it is first accessed. The implementation is listed in Figure 28.

The lazy initialization can not be replaced by automatic initialization. It can not ensure a singleton instance of class LOCALTRANSLATOR through setting up the accessibility of the member variable singleton as global or static and initializing it at the declaration time or in constructor. First, it can not be guaranteed that only one

instance of a static object will be declared. Second, the initialization of a singleton may rely on other runtime values and thus it can not be sure that a singleton can be initialized using static initialization with required information.

4.3.3.3 Implementing Flyweight

Effectiveness is achieved through the design that runtime data structures are organized using *Flyweight Design Pattern.* However, it also brings up costs of finding shared objects and computing required information. So managing shared objects efficiently is critical to make the design practical.

The flyweights, or the shared nodes, should be stocked with little effort. Besides, the storage of them should facilitate locating a particular object. Generally, an association will be used to look up interested flyweights. The data structure of hashtable is suitable for the purpose. All the flyweights compose a table. In this table each flyweight is stored and retrieved using its key, the node ID. Also, an edge or a blob refers to a flyweight by having the key of the flyweight in its state. The edge or the blob only know the identities of flyweights. To obtain further information of its related flyweights and to modify the inner states of them, it will retrieve the flyweights and compute. Note that if a flyweight is altered during the computation, it need to be restored to the repository table.

4.3.3.4 Implementing Coral Program Generation

A syntax error free CORAL program can not guarantee that the execution yields expected results. Not only the logic behind the CORAL program should be proper, but the constraints of the CORAL program must be satisfied. These increase the complexity of the implementation of generating a health CORAL program.

At the current version, the CORAL system has the following constraints that relates to Increment One:

- A rule need to have its variables bound before being evaluated. Still using the example in Figure 26 to illustrate the constraint. Suppose the rule $creditMT2(Code) :- course(Code, \_, Credits), Credit > 2.$
  is changed to $creditMT2(Code) : -Credit > 2,\ course(Code, \_, Credits).$,
  there will be no answers at all although several course Codes are derived using the first program. The reason behind the scene is that expression $Credits > 2$

is evaluated first, and at the very time, the variable Credits has no value at all. Even immediately later, Credits does have a value through a fact of the predicate course, the interpreter of CORAL system will not check back.

Through the following implementation strategies, the constraint is guaranteed to be met.

- In the rule body, list predicates for nodes before those of edges, and lastly those for blobs. The rationale of the strategy lies in simple components composing complex ones. Since an edge consists of two nodes, then listing nodes' predicates before the edges' will ensure variables, which appear in node relations but are used by edges, will be bound before referred by edges. Since blobs are defined in terms of nodes and probably edges, the same analysis applies.

- List comparison expressions after system predicates and user-defined predicates. This is because the variables in the comparison expressions exclusively refer to variables that appears in those predicates. The arrangements can prevent a similar error in the example above.

4.3.4 Increment One — Test

Use case testing method [11] is adopted to examine the application functionality. Use case testing is designing, executing, and evaluating test cases directed by use cases. Because the translation system is independent of user interface, the test will proceed without involving the user interface. However, since most test inputs have the format of XML file, they are significantly abstract. This also causes the evaluation of the test result to be difficult. For a good understanding of the inputs , the corresponding drawing patterns are drawn together with generated TGL program.

Each test case is designed to be self-supported. Its TGL program is designed to conforms to the corresponding use case inputs. After the test completes, the post-conditions of the use case will be validated. The independence of the test cases has the following benefits:

- All test cases can be used repeatedly. Because the system environment need to be set for the test, satisfying post-conditions of one run can not verify the correctness of the application functions. Many runs must yield the same results. An independent test case can be used for this purpose.

- They do not intervene with each other. The inputs and pre-conditions of a test case are not dependent on the post-conditions of others, and the results of test cases are isolated as well. Thus, the failure of a few tested cases can not derive the incorrectness of the rest of the cases.

- Standalone test cases prevent complicated setup. Because the results of some test cases are not the necessary preparation for others, the test cases are not chained. This precludes the need for ensuring that each preceded tests yields expected production. This causes extensive time to set up a single test.

- Separate test cases support efficient debugging. The debugging can be carrying on from case to case. Therefore, the complexity is reduced and debug speed can be faster. As a result, efficiency is achieved.

The test cases are developed using the same implementation language, Java, as the translation system.

The upper layer system expects no human user assistance during the interactions with the translation system. Besides, it expects as few as possible interaction to attain goals. To fulfill this requirement, simple interfaces are demanded from the translation system. This results in a significant information-hiding design effect of the translation system. Nonetheless, the effect draws on the difficulty of validating test results. The tester will only have file names of results, which is far from enough to verify the functionality of the translation system. To resolve the problem, in the test phases, intermediate products of the translation process are held and traced so that detailed examination could be carried on.

The university model is taken on for the test purpose. The reason why we use the university model is that university domain is well understood and many database systems related to this domain have been established. Consequently, detailed problems can be discussed with common understanding, complex test cases can be designed and followed without causing confusion, and horizontal comparsions can easily be set up with other system.

In Increment One, we listed a test case as an example to demonstrate that the design and implementation for Increment One could achieve the functions required by use cases. Also this example is used to exhibit the effectiveness of the use case testing method.

4.3.4.1 Test Case — Fetch System Schemes

The test case named Fetch System Schemes is based on the use case fetchSystemSchemas. It uses the file systemScheme.txt, which is updated through initializing the translation system. It can also be assumed that the file systemScheme.txt will not change before the translation system shuts down and reinitializes.

Input Explained The schemes recorded in systemScheme.txt are supposed to be retrieved and organized from a relational database. This is the routine of a deductive system. Every line in the table represents a relation. The relation name is listed first, then attributes are listed in the following parentheses. The order of the attributes is important because a deductive database treats attributes by their positions instead of their names. In the example input shown in Table 9, the relations of lives_in, first_supervisor, second_supervisor, prerequisites, assessment, resides assessment reflect relationships between two entities. The rest are of entity model. Whereas the relations representing relationship will be depicted with edges or blobs, an entity relation will correspond to a node in the TGL, and the first attribute should be its key. Giving an example, the predicate *prerequisites* of assessment reflects a relationship between a course and its prerequisite courses.

Test Object The test object developed for this test case are listed in Figure 29. This test case is a simple one , and only one *main* method is enough. Nonetheless, for complex ones, more methods need to be created to assist the tests.

Test Execution and Evaluation Under the conditions:

- Underlying layer system could be connected.
- Relation schemes can be retrieved through the underlying system and systemSchemas.txt can be updated.
- There indeed exist schemes like in Table 9.

The test result copied from console is listed in Figure 30.

person(ID, Name).
staff(ID,Salary).
student(ID).
tutor(ID).
visiting_staff(ID).
dept(No, Name).
course(Code, Title, Credit).
address(AID, Street, District, City).
works_in(ID, Dept).
teaches(ID, Course).
majors_in(ID, Dept).
takes(ID,Course).
run_by(Course,Dept).
prerequisites(Course,PreCourse).
assessment(Course,assName,Percent).
lives_in(ID, AID).
first_supervisor(StaffID, StudentID).
second_supervisor(StaffID, StudentID).
resides(Dept, AID).

Table 9: The System Scheme File shared with lower layer system

```
public static main(String[] args) {

  System.out.println("Date: " + new Date());
  System.out.println("Test Case --- Fetch System Schemas");
  // precondition set up \: initialize the translator
  LocalTranslator translator = new LocalTranslator();
  try{
   int updateStatus = translator.initialize();
   if(updateStatus==translator.NotUpdated){
       System.out.println("underlying system not ready.");
       System.exit(1);
    }
  }catch(FileNotFoundException e){
   System.out.println("Initializing Failure ...");
   System.exit(1);
  }

  // retrieve the system schemas and show results.
  try{
  Hashtable systems = translator.fetchSystem();
  catch(
  int count = 0;
  for(Enumeration e=systems.keys(); keys.hasMoreElements();){
   count++;
   String name = (String)keys.nextElement();
   String[] fields = (String[])systems.get(name);
   System.out.println("Scheme " + count + " : " +name);
    int k = 0;
    for(int k = 0; k<fields.length; k++){
            if(k == fields.length-1){
              System.out.print(" " +fields[j] + "\n");
            }else{
              System.out.print(" " + fields[j] +",");
            }
    }
  }
  if(count==0){
   System.out.println("No existing system schemas available.");
   System.out.exit(1);
  }else{
   System.out.println("There are together " + count + " system schemas.");
  }
}
```

Figure 29: Test Object for Fetch System Schemes

```
lice.l_zou % java TestOneSystem
Date: Thu Dec 12 20:10:34 EST 2002
Test Case --- Fetch System Schemas
Scheme 1 : person
 ID,Name
Scheme 2 : staff
 ID,Salary
Scheme 3 : student
ID
Scheme 4 : tutor
ID
Scheme 5 : visiting_staff
ID
Scheme 6 : dept
 No,Name
Scheme 7 : course
 Code, Title,Credit
Scheme 8 : address
 AID, Street, District,City
Scheme 9 : works_in
 ID,AID
Scheme 10 : teaches
 ID,Dept
Scheme 11 : majors_in
 ID,Course
Scheme 12 : takes
 ID,Dept
Scheme 13 : run_by
 ID,Course
Scheme 14 : prerequisites
 Course,Dept
Scheme 15 : assessment
 Course,PreCourse
Scheme 16 : lives_in
 Course, assName,Percent
Scheme 17 : first_supervisor
 StaffID,StudentID
Scheme 18 : second_supervisor
 StaffID,StudentID
Scheme 19 : resides
 Dept,AID
There are together 19 System Schemas.
```

Figure 30: Test case result of Fetch System Schemes

4.4 Increment Two Process

This section illustrates how Increment Two integrates the core functions into the translation system. Still the waterfall method is used to go through the process of analysis, design, implementation, and test of these functions. Besides the joining of new contents, updates are made to the deliverables of Increment One to meet new requirements. However, the modifications to the existing desing and implementation only limit increasing class attributes and methods, but no associations between classes are altered.

Increment Two focuses on how to record queries, transform queries to CORAL programs, and make use of CORAL system to execute the programs,

4.4.1 Increment Two — Analysis

This section analyzes the requirements for retrieving query results through the translation system. The deliverables of Increment One support the analysis in Increment Two by having a similar process, from receiving the user requests to transforming into executable CORAL programs. However, it has to go a step further to complete a query instead of defining a relation. The generated CORAL program that corresponds to a TGL program needs to be submitted to the underlying CORAL system through runtime environment and get executed. Furthermore, the execution results of the CORAL programs need to be checked and transformed into TGL programs. At last, the transformed results will be sent back to the upper layer system.

4.4.1.1 Use Cases refinement

This part refines use cases in Increment Two.

Use case	queryEntityReferingToOnlySystemSchemas
Scenario 1	The upper layer system submits the TGL programs that query one or many entity relations to the translation system. The translation system parse the program and generate corresponding CORAL programs. Then it invokes the CORAL system through the enclosed operating system. Then after the CORAL system executing the programs, the translation system collects the results and transformed them into TGL program and inform the upper layer system.
Pre-condition	The translation system is ready, and the CORAL system is reachable through the runtime environment.
Post-condition	a set of TGL programs corresponds to the CORAL execution results.
Exception	The CORAL system is not available. I/O exceptions during reading and writing files. The translation process halted because of the illegalities of the TGL program.
Use case	queryEdgeReferingOnlyToSystemSchemas
Scenario	After taking over TGL programs for querying about edge relation(relations), the translation system analyzes them and generates CORAL programs accordingly. Then it connects to the CORAL system and requires the execution. With the execution results, it translates them into the format that the upper layer system can recognize, then transfers them back.
Pre-condition	TGL programs including the information of the distinguished edge(edges) and its context.
Post-condition	Query results for the distinguished edge(s) in the format of TGL programs.
Exception	TGL programs are not acceptable. Input/Output problems caused by environment. CORAL system is unavailable.

Use case	queryBlobReferingToOnlySystemSchemas
Scenario	With the TGL program that aims to find pairs of container node and contained nodes, the translation system checks its syntax and translates the semantics. The newly generated CORAL programs are executed by the CORAL system after successful connections. Then the translation system takes back the results, and transforms them in the unit of blob instances. Last the regenerated results are sent back to the upper layer system.
Pre-condition	legal TGL query programs that emphasize on searching instances of existing blob(s).
Post-condition	Blob instances found in the format of TGL programs.
Exception	The blob to query does not exists. TGL programs has logical errors.
Use case	queryEntityReferingToUserDefined
Scenario	In the TGL program that user submitted, entity relation results are interested. Moreover, user defined relation(s) are in the context to support the query. So besides the normal translation, it needs to find all related user defined relations recursively. And before executing the programs through the CORAL system, it should consult all these relations. Then it translates the execution results into TGL programs.
Pre-condition	Legal TGL programs, accessible run-time data structures of user defined relations and their direct supporters.
Post-condition	instances of interested entity relation(s) retrieved and in the format of TGL programs.
Exception	No accessible information of user defined relations and their supporters. The user-defined relation referred to has no corresponding CORAL programs defined.

Use case	queryEdgeReferingToUserDefined
Scenario	The upper layer system transfers a TGL program that interests the instances of edge relation based on user-defined relations. The translation system collects all the direct and indirect user-defined relations besides translation. Then it first consults these user-defined relations before consults generated CORAL programs. Last, the translation system transformed the received CORAL execution results into TGL programs and informs the upper layer system.
Pre-condition	Legal TGL programs, information of user defined relations and their direct supporters.
Post-condition	Results in the format of TGL program that corresponds to the CORAL execution results.
Exception	The user-define relations referenced has no information either for the translation system or for the CORAL system.
Use case	queryBlobReferingToUserDefined
Scenario	TGL programs that aim to find instances of interested blob(s) are handed over to the translation system. Along with the generation of the CORAL programs, the translation system also search all the direct or indirect user-defined relations. Then before executing the CORAL programs generated, it consults all the referred user-define relations in the order of their dependencies. At last, it receives the results and makes translation again before sending back to the upper layer system.
Pre-condition	TGL programs including highlighted blobs to be queried and names of referred user-defined relations.
Post-condition	A set of blob instances the met the query pattern.
Exception	No useful user-defined relation information.

4.4.1.2 Analysis Class Diagram

The class diagram shown in Figure 31 is based on the use cases in Increment Two. Two more classes are added and referred by XENGINE: XSHOWGRAPHLOG and

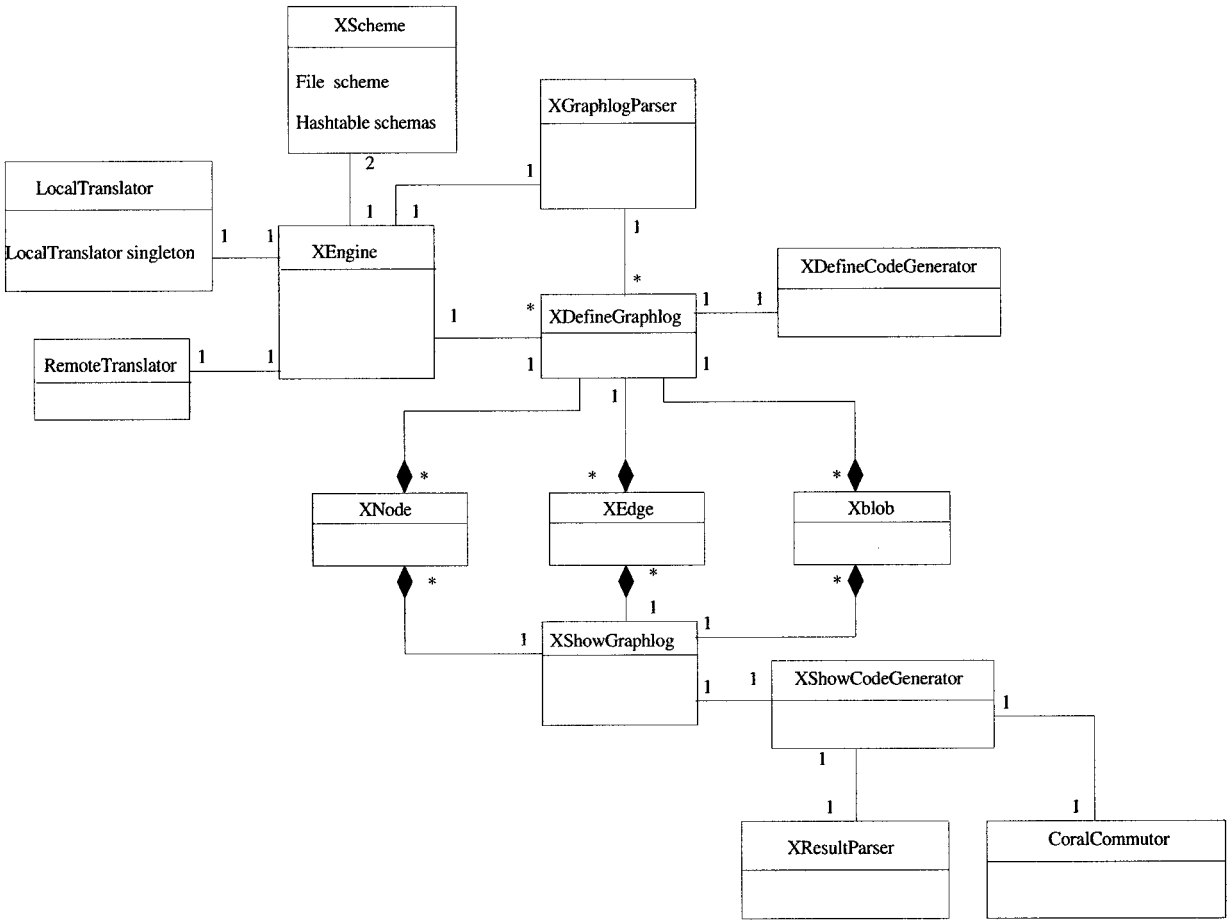


Figure 31: The Analysis Class Diagram of Increment Two

XSHOWCODEGENERATOR.

Note that the naming convention for the classes in engine package are prefixed with the character 'X', this is used to differentiate from classes with similar names in other packages.

4.4.2 Increment Two — Design

4.4.2.1 Conceptual Structure of Increment Two

At this stage the structure for use cases in Increment Two is augmented based on that of Increment One and shown in Figure 32. In the next increment, the structure will still be augmented.

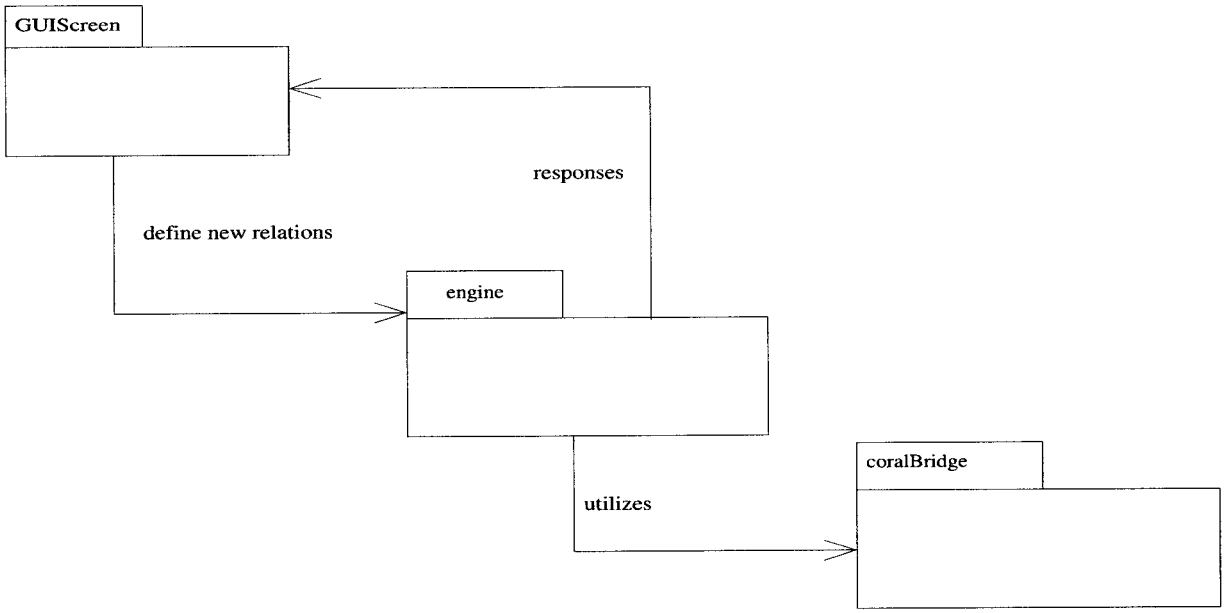


Figure 32: The conceptual structure at stage of Increment Two

4.4.2.2 Updated Inner Class XRuntime

In Increment two, a new class XSHOWGRAPHLOG is introduced to deal with the showGraphlogs. And showGraphlogs have great similarity with defineGraphlogs in terms of refining run-time data structures. unlike defineGraphlogs, showGraphlogs deal with the utilizing of included user-defined predicates. DefineGraphlogs simply register the included. But showGraphlogs must list all referred user-defined predicates, including those indirectly refereed predicates. So the run-time refinement for showGraphlogs involves refining the corresponding data structure as well.

Therefore, the inner class XRUNTIME, which is responsible for refining data structures of XGRAPHLOG, is updated as Figure 33 illustrates. The newly added methods are in bold face.

4.4.2.3 Transforming showGraphlogs to CORAL Programs

There are similarities between transforming defineGraphlogs and transforming showGraphlogs. Nonetheless, the differences do exist.

First, a showGraphlog may own a bunch of distinguished elements while a defineGraphlog can only have one distinguished element, whatever it is, a node, an edge, or a blob. So , for a showGraphlog, all the distinguished elements need to be checked

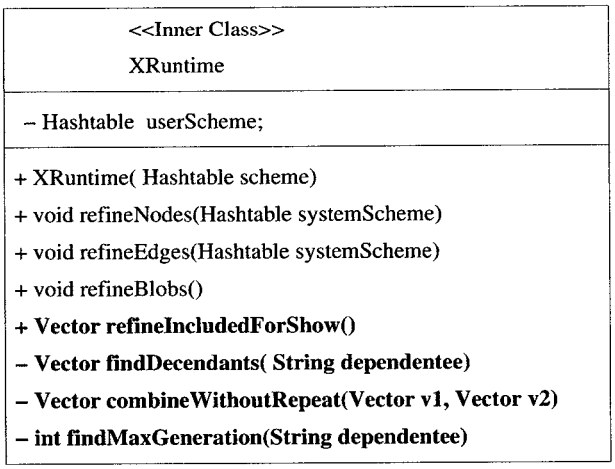


Figure 33: Updated Inner Class XRuntime

carefully to locate all interested query objectives instead of just one for a defineGraphlog. Since the purpose is to find the possible values for the objectives, a single predicate could be employed to encompass as attributes all the interest objectives.

Second, because a showGraphlog aims at finding the result, the predicate it defines is simply an intermediate product. Thus, unlike a defineGraphlog, a showGraphlog does not keep the newly declared predicate permanent, and the predicate is discarded after usage.

Third, the interested objectives lose the connection with their origins, the nodes, edges, and blobs when they are simply dug out. This does not influence retrieving results at all since the context is also woven into the CORAL program as rule bodies. However, when the query results come out, the objectives with many values suddenly find they are orphans, at least they can not find any clue of their own family, the nodes, edges, and blobs. Furthermore, They are not in a format that is acceptable for the upper layer system who is expecting them. The solution to this dilemma is that along with the process of picking the objectives, the code generator also records their origins and the roles they play in the origins. Later, with this data structure, the query results derived from the CORAL system can be reorganized into the TGL programs understood by the client.

4.4.2.4 Accessing CORAL System in Background

ShowGraphlogs declare the query patterns to retrieve query results. And processing queries needs the assistance of the CORAL System.

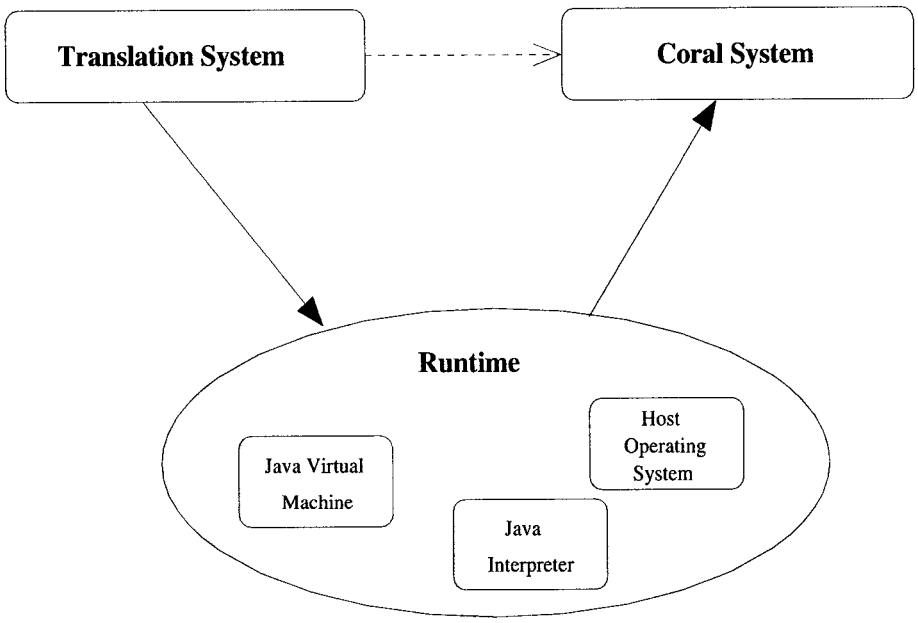


Figure 34: Accessing CORAL System

For a particular showGraphlog, not only the transformed CORAL program, but the set of user-defined relations that are referred by the showGraphlog are consulted in advance to attain the results. Although CORAL System encourages interacting mode of queries for users, it allows background execution, too. There are two key points about the execution of generated CORAL programs from a showGraphlog: messaging CORAL system and executing CORAL program in background.

Java Runtime Environment and Messaging CORAL System To send messages CORAL System is in fact the problem of calling CORAL system through Java programs under the support of the underlying platform. The environment where the translation system sits has determined influence to the solution of the problem.

The translation system resides at the UNIX platform [18] with the support of UNIX version CORAL system. But the translation system can not see the CORAL system although both of them live in the runtime environment of UNIX system. There has to be a way to let the translation system to first access the runtime environment and then to talk to the CORAL system. A RUNTIME object of the Java environment is employed as the bridge. The RUNTIME object allows the application to send messages to the system resources directly. The process is illustrated in Figure 34.

The oval labelled Runtime in the diagram represents the current runtime environment and is an instance of the Runtime class. The current runtime environment could be an implementation of the Java virtual machine and an interpreter running on host operating system, or an implementation of the virtual machine together with an interpreter on a particular operating system.

RUNTIME objects provide two services [23]. First, they communicate with the components of the runtime environment—getting information and invoking functions. Second, RUNTIME objects are also the interface to system-dependent capabilities. For example, UNIX Runtime objects might support the getenv and setenv functions.

Nonetheless, accessing directly to the system resources through a RUNTIME object does have a disadvantage: it comprises the ability of system-independence because the Runtime class is tightly integrated with the implementation of the Java interpreter, the Java virtual machine, and the host operating system. So the translation system is system-dependent at this point.

Execute CORAL programs at backend Since the translation system can summon the underlying UNIX operating system, the CORAL system can be called in turn using the command "coral" to enter into the interaction mode to execute queries. However, the interaction mode requires manual involvement, which is inflexible and impractical for the translation stem.

CORAL system does provide a way to execute programs at the backend [6]. The format of the command is: $coral < inputFileName > outputFileName;$. Here the keyword *coral* is the UNIX command to run the CORAL system. The symbols ' $<$' and ' $>$' introduce an input file and an output file respectively. Last, the semicolon marks the end of this command. The inputFileName and outputFileName are provided by users. In the input file, the CORAL commands for processing queries are listed; and in the output file, the CORAL system feeds back the results into it. An example of output file is listed in Figure 35.

To execute a list of the commands at the backend, the user lists them in an executable file. The execution of the file will run the commands sequentially. Another line, *#!bin/bash*, must exist at the very top of the file because the commands conforms to the bash script language.

```
This is Coral Version 1.5.2
-----------------------------------------------------

    Welcome to CORAL.

    All commands MUST end with a period .
    Type help. to access help information.

-----------------------------------------------------

ready>>ready>>ID=4881177.
... next answer ? (y/n/all)[y]ID=4125785.
... next answer ? (y/n/all)[y](Number of Answers = 2)
ready>>
```

Figure 35: An example of CORAL Backend Output

Organizing the Access Class CORALCOMMUTOR is the coordinator of the whole process of accessing UNIX operating system, communicating with CORAL, executing CORAL programs, and messaging back the execution results. It and the executable file *executeCORAL*, the input file *inputFile.txt*, and the output file *outFile.txt* all belong to the package of *communicateCORAL*. They collaborate together with XShow-Graphlog to accomplish the process. And the relationship is demonstrated in Figure 36

4.4.2.5 Translating CORAL Execution Results to TGL program

Like Figure 35 shows, the results obtained through the CORAL system are listed for variable attributes of the predicate. And these are bound values from derived facts for the predicate. However, the upper layer system is not able to understand them and feeds them back to the human users. So another circle of translation is required.

STRINGTOKENIZER **Class** No new tokenizer class is designed for scanning the result file, the STRINGTOKENIZER class in Java class libraries has enough functions to accomplish the job.

Another BUFFEREDREADER class of Java class libraries aids in the scanning process. A BUFFEREDREADER object swallows the whole result file, but spits it out line by line. A STRINGTOKENIZER object is then fed with selected lines that includes the answers, but not decorations. For each line, the STRINGTOKENIZER object strips out CORAL prompt and explanation. Then it breaks the rest to variable and value

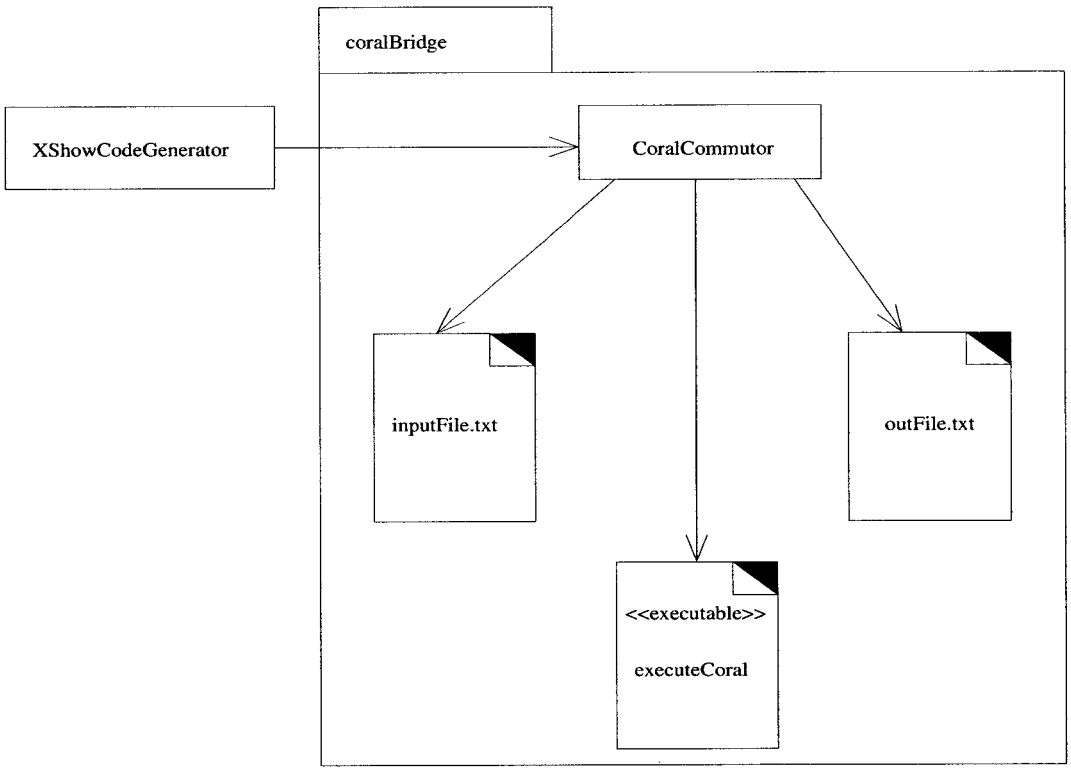


Figure 36: Collaborating to Access the CORAL System

pairs.

XRESULTPARSER **Organizing Translation** Unlike the XPARSERS in Increment One, which are responsible for recognizing TGL patterns and building run-time data structure. The XRESULTPARSER class takes care of the whole process of translating the result file.

It takes the above-mentioned BUFFEREDREADER object and STRINGTOKENIZER object as its members to accomplish the scanning job. And it creates the final result file for the upper layer system with the help of two HASHTABLE objects. First, an XRESULTPARSER object is constructed by receiving the file to be translated. In turn, it creates the BUFFEREDREADER object by passing the file continuously. However, the XRESULTPARSER class differentiates from a general parser that recognizes patterns after a scanner tokenizing the input stream. Before requiring the STRINGTOKENIZER to decode, it filters out what the STRINGTOKENIZER should work on. Next, with the two HASHTABLE, it constructs results according to the result schema regulated in TGL. The two HASHTABLE objects are essentially two views of the same

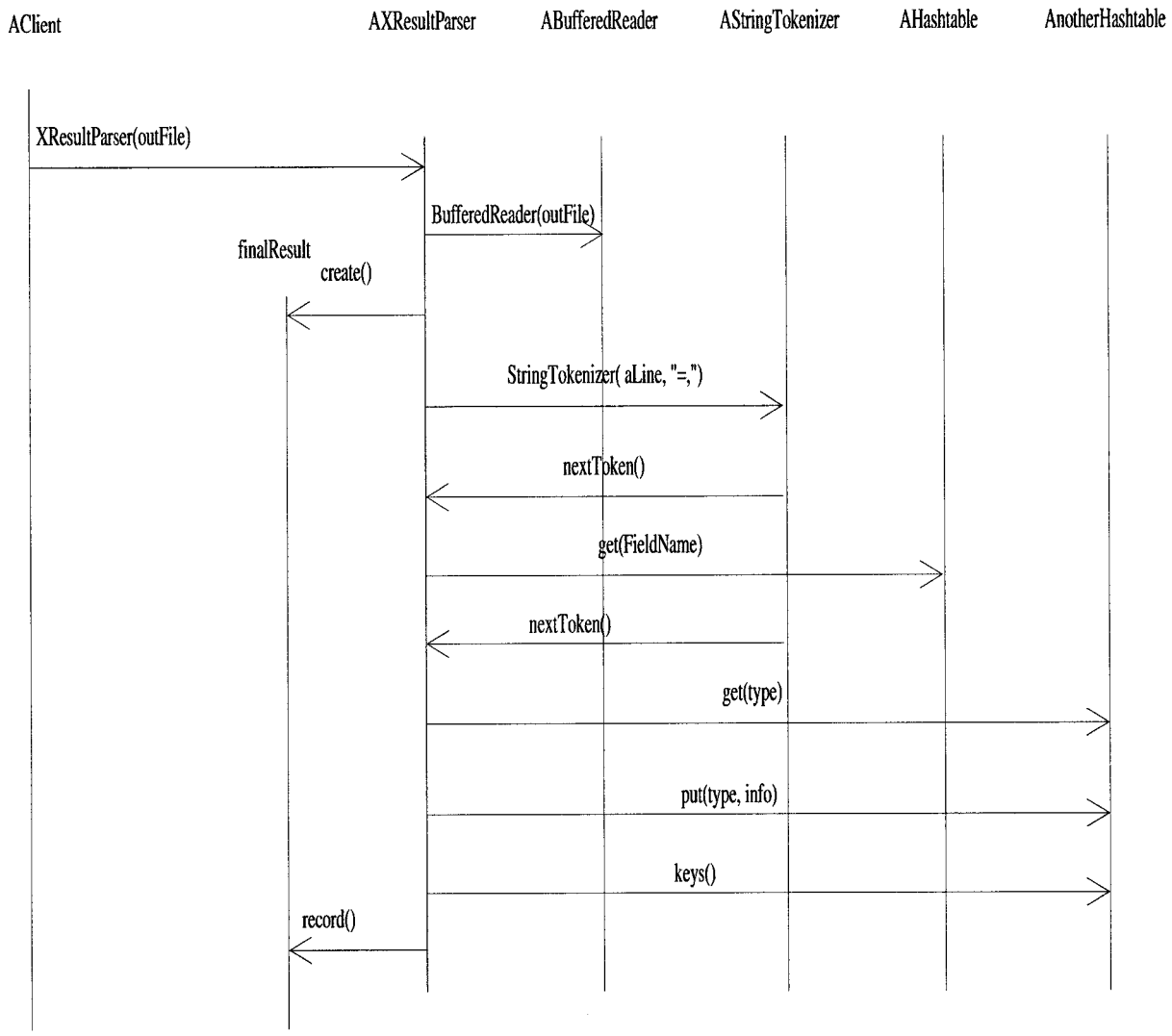


Figure 37: The Sequence Diagram for Parsing Result

data. One HASHTABLE object is built up in a XSHOWCODEGENERATOR. It has the variables to be queried as index, and lists the associated distinguished nodes, edges, or blobs and the roles played of variables. The other HASHTABLE object sees the data of the first Hashtable object in the unit of node, edge, and blob. The second HASHTABLE object collects all its indices while dealing with first line answers. And the remaining lines share the HASHTABLE object structure and change the contents for keys. After toknenizing each line, a unit answer is formed in terms of result schema and output to the final result file. The sequence diagram shown in Figure 37 illustrates the collaboration.

The design delivery of Increment Two is illustrated in Figure 38.

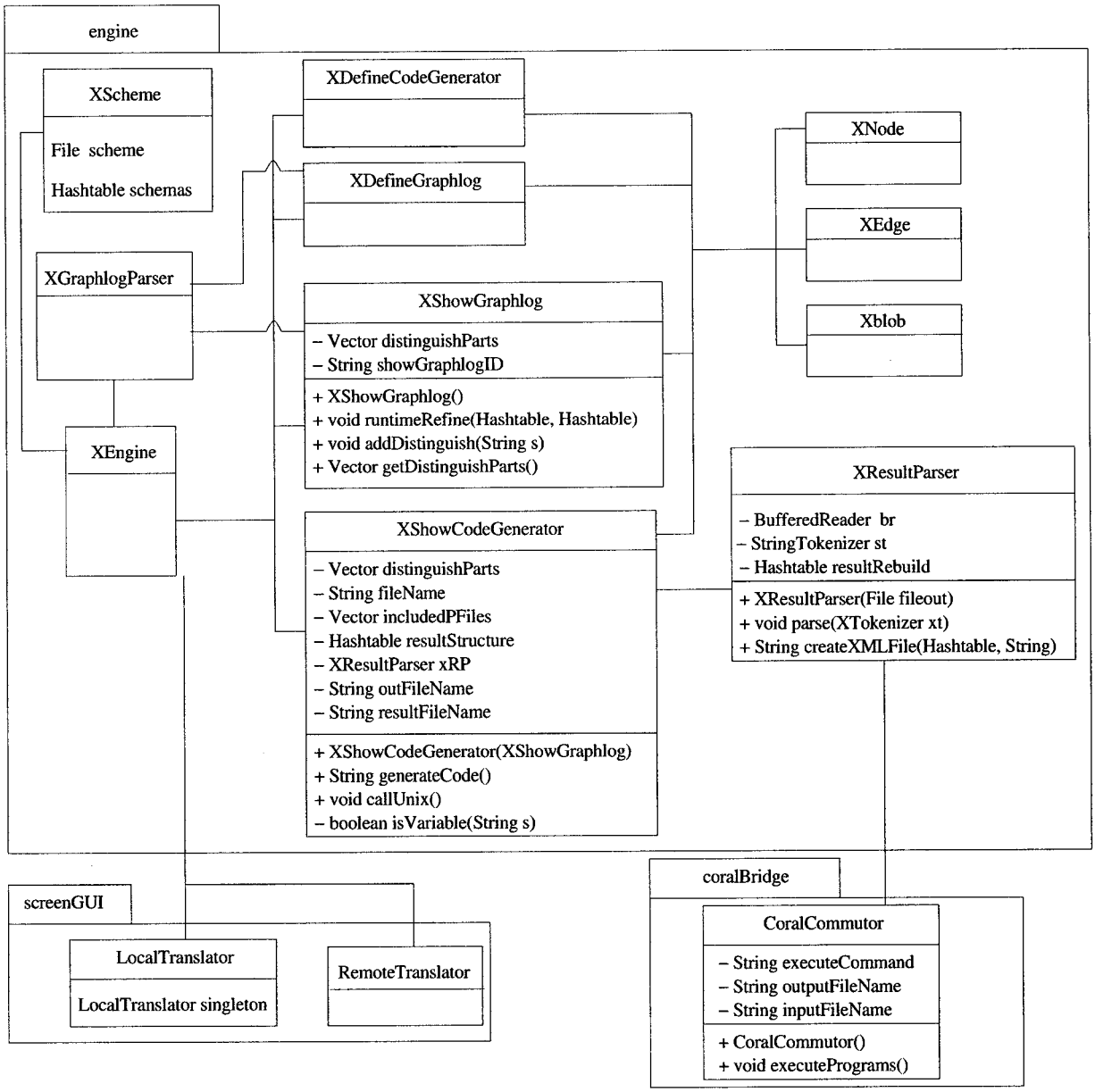


Figure 38: The design class diagram of Increment Two

4.4.3 Increment Two — Implementation

4.4.3.1 Finding All Used User-Defined Relations of ShowGraphlogs and Recursive Method

This part also constitutes the implementation of newly designed methods of inner class XRuntime in increment two.

For a showGraphlog, a list of user-defined relations may be referred to. The definition of such a relation, say A, may rely on not only system schemes but also other user-defined relations, say B, C, etc. In turn, the B and C may still depend on other user-defined relations.

CORAL system requires that the definition of a predicate should be provided before it is utilized. So all recursive used definitions should be listed before those of included user-defined predicates. Consequently, for a showGraphlog, besides other runtime refinement, it needs to look up and reorganize the information of all actually used user-defined schemes, which are a superset of the included files listed in the corresponding TGL program.

Two steps are involved in the solution. First, it must find both directly and indirectly referred user-definition shemes. The implementation makes use of recursive methods. For each listed included user-defines relation A in the TGL program for a showgraphlog, it must record its utilized user-defined relations, which is a set T. In turn, for each relation in the set T, the process continues and new records are collected without repetition. The process ends when no more new user-defined relations can be found. The code segments are listed in Figure 39 and Figure 40. The end point of the recursive method exists because the dependency relationships among user-defined predicates are acyclic. It could be proved because it is impossible that A's definition uses B's definition , and at the same time, B's definition depends on A's definition. Notice that the method *combineWithoutRepeat* is a utility to ensure that the definition of a single user-defined predicate is not added twice in the execution.

Second, it must satisfy the constraint that the definitions of predicates that are depended on always stands before those of its dependents.

A fact exists that if two predicates have the same number of descendant generations, it is not possible that one could depend on the other. *The Number of Decendant Generation* is defined as follows: If the definition of an ancestor relies on the definitions of some other predicates, which are called the next generation of the ancestor,

```
Vector findDecendants(String dependentee){
    Vector decendants = new Vector();
    Vector nextGeneration = (Vector)userScheme.get(dependentee);
    if(nextGeneration == null){ // no next generation
        decendants.add(dependentee);
    }else{
        for(Enumeration e = nextGeneration.elements(); e.hasMoreElements();){
        String aDependant = (String)e.NextElement();
        Vector aGeneration = findDecendants(aDependant);
            combineWithoutRepeat(decendants, aGeneration);
        }
        if(!decendants.contains(dependentee))
            decendants.add(dependentee);
    }
    return decendants;
}
```

Figure 39: Implementation of Recursively Recording Related User-defined Relations

```
Vector combineWithoutRepeat(Vector v1, Vector v2){
        if( v1==null && v2 == null)
          return null;
        if(v1 == null)
          return v2;
        if(v2 == null)
          return v1;
        Vector combination = new Vector();
        combination.addAll(v1);
        for(Enumeration e = v2.elements(); e.hasMoreElements();){
            Object anElement = e.nextElement();
            if(!combination.contains(anElement)){
                combination.add(anElement);
            }
        }
        return combination;
}
```

Figure 40: Implementation of Combining Collection without Repetition

```
int findMaxGenerations(String dependentee){
    Vector nextGeneration = (Vector)(userScheme.get(dependentee));
    if(nextGeneration == null){
            return 0;
    }else{
      int max = 0;
      for(Enumeration e = nextGeneration.elements(); e.hasMoreElements();){
            String aDependentee = (String)e.nextElement();
            int generations = findMaxGeneration(aDependentee);
            if(max<generations)
                max = generations;
      }
      return max;
    }
}
```

Figure 41: Implementation of Calculating Number of Generation

but none of the next generation defines referred to any user-defined relations, then the ancestor has one descendant generation. And its descendants are all leaves. However, if any descendants of the ancestor also has their own descendants, then the number of generations of the ancestor is one plus the maximum number of generations of its dependants.

The fact can be proved. First, we assume the fact does not stand. Thus we suppose predicates A and B have same number of generations N, but A is the ancestor of B. The conflict can be derived as such: since A is the ancestor of B, then according to the above definition for *The Number of Decendant generation*, at least, the number of descendant generation of A is one plus that of the B, that is, $N+1$. As a consequence, the conclusion disagrees with the assumption of A's number of generation being N. So our assumption can not hold, and the fact is true.Based on the fact, we could ensure any ancestor is listed before its descendants by the following strategy: compute all the number of generations for all derived user-defined schemas in the first step. Then list the derived schemes according to their number of generations, with the order from the lowest to the highest. The code for the calculation of the number of generations is in Figure 41.Note that he data structure hashtable is employed to keep the records of relations given a number of generations. The corresponding implementation code for the whole procedure is in the Figure 42.

```
Vector recursiveInclus = null;
int maxLevel = 0;
for(Enumeration e = includedPFiles.elements(); e.hasMoreElements();){
  String aPFile = (String)e.nextElement();
  Vector aFamily = findDecendants(aPFile);
  combineWithoutRepeat(recursiveInclus, aFamily);
}

Hashtable pFilesWithLevel = new hashtable();
for(Enumeration e = recursiveInclus.elements(); e.hasMoreElements();){

      String aPFlie = (String)e.nextElement();
      int levels = findMaxGenerations(aPFile);
      if(maxLevel < levels)
         maxLevel = levels;
      Vector pFilesAtSameLevel = (Vector)( pFilesWithLevel.get(Integer.toString(levels)));
      if(pFilesAtSameLevel == null){
        pFilesAtSameLevel = new Vector();
        pFilesAtSameLevel.add(aPFile);
        pFilesWithLevel.put(Integer.toString(levels), pFilesAtSameLevel);
      }else{ // the vector for this level has existed.
        pFilesAtSameLevel.add(aPFile);
        pFilesWithLevel.put(Integer.toString(levels), pFilesAtSameLevel);
      }
}
 Vector organizedInclus = new Vector();
 for(int i =0; i<=maxLevel; i++){
      Vector aLevel = pFilesWithLevel.get(Integer.toString(i));
      if(aLevel != null){
           organizedInclus.addAll(aLevel);
      }
 }
```

Figure 42: Implementation of Listing All Based User Schemes In order

Because these procedures serve to refine the runtime data structure of showGraphlogs, characteristically they belong to the class XRuntime. So the above implementation is added to the inner class XRUNTIME of XGRAPHLOGPATTERN as member methods.

4.4.3.2 Messaging CORAL System Using a Runtime Object

As any other Java applications, the translation system has a single instance of class RUNTIME that allows it to interface with the environment in which it is running. However, like others, the translation system cannot create its own instance of this class. The way to obtain its runtime is using the *getRuntime* method of RUNTIME class: Runtime aRuntime = Runtime.getRuntime(). Then, with the aRuntime, the translation system could start the CORAL system.

4.4.4 Increment Two — Test

This part describes the test of the deliverables of Increment Two. The test cases are designed in accordance with the user cases listed for the current stage and center on processing queries through the translation system given legal TGL programs.

All the scenarios related to the use cases for Increment Two are tested. Nonetheless, only one typical scenario as well as its test results and intermediate products are explained in detail as follows.

4.4.4.1 The Query Example Graph

In the university data model, the query *Return staff members and students in the computing science department* could be expressed as in Figure 43

In the figure, the upper rectangle first defines a relation named dept_people using a blob. All the underlying schemes used are system schemes. The schemes could be found in Table 9. The label of blob is dept_people. The container node is an entity with relation dept. There are two contained nodes for this blob, and they are staff entity and student entity.

The lower rectangle declares the query for retrieving final results. The query is based on the newly defined relation dept_people in the upper rectangle. Two kinds of entities, staff and student, are interested. Note that another attribute of the relation

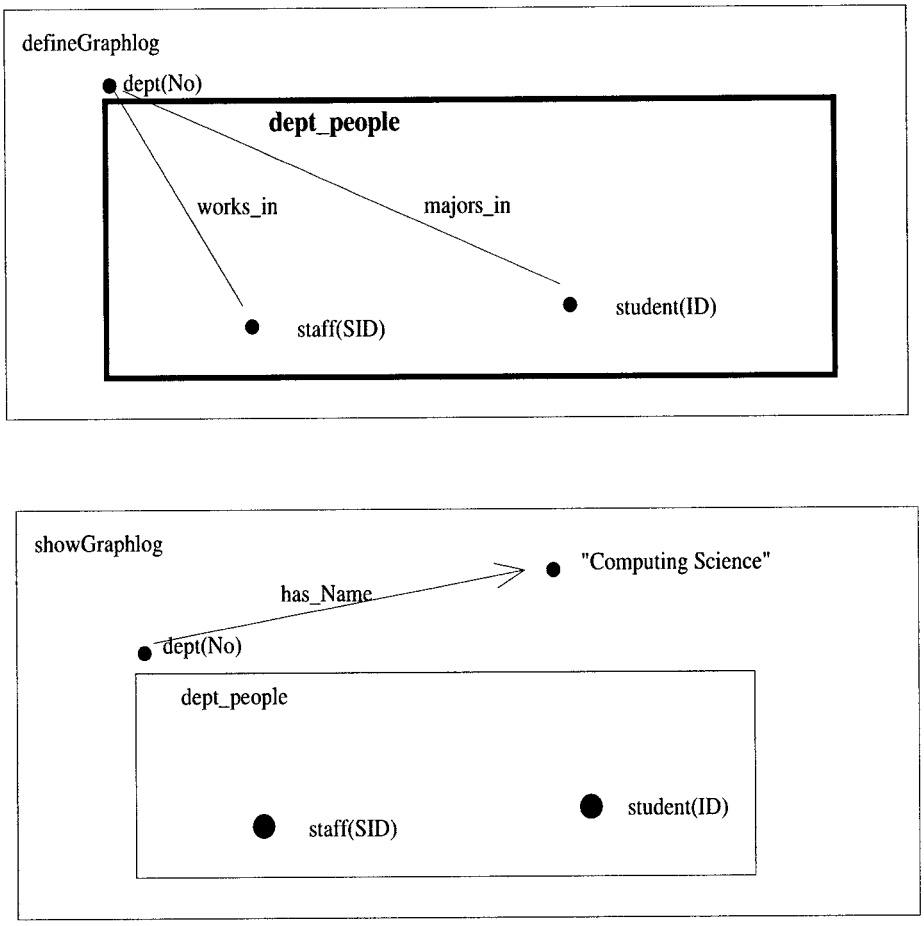


Figure 43: Drawing of Query "Return Staff Members and Students in the Computing Science Department"

```
module dept_people.
export dept_people(ff).
eid002(ID,No) :- majors_in(ID,No).
eid001(SID,No) :- works_in(SID,No).
dept_people ( No , SID ) :-
eid002(ID,No),
eid001(SID,No).
dept_people ( No , ID ) :-
eid002(ID,No),
eid001(SID,No).
end_module.
```

Table 10: The CORAL Program That Define Relation dept_people

dept is depicted through the edge labeling *has_Name*. As designed above, it provides flexibility to describe the *Name* attribute so the value, "Computing Science", can be emphasized. In addition, it gives a visual clue of the relationship between the entity *dept* and its attribute *Name*.

Note that two entity relations, student and staff, are interested because the query aims to distinguish retrieved instances of them. Otherwise, if the query only want people in Computing Science department, only one node can be drawn instead of two.

4.4.4.2 Transformed TGL Program

The transformed TGL program that corresponds to the pictures are listed in Figure 44.

4.4.4.3 Generated CORAL Program

The translation system parses this program and generated a user-defined relation dept_people is shown in Table 10

4.4.4.4 CORAL Execution Results

Figure 45 displays the execution results of the CORAL system.

It is obvious that there are repetitions among the results for a single relation. This is because of the way the query pattern is expressed.

```
— Return students taking a course given by Steve Johnson
<graphlog>
 <defineGraphlog>
  <distinguished–define>
   <blob>
    <ID>BID001</ID>
    <predicate>dept_people</predicate>
    <outerNodeID>NID0001</outerNodeID>
    <innerNodeID>NID0002</innerNodeID>
    <innerNodeID>NID0003</innerNodeID>
   </blob>
  </distinguished–define>
  <content>
   <node>
    <ID>NID0001</ID>
    <entity>
     <name>dept</name>
     <field>No</field>
    </entity>
   </node>
   <node>
    <ID>NID0002</ID>
    <entity>
     <name>staff</name>
     <field>SID</field>
    </entity>
   </node>
   <node>
    <ID>NID0003</ID>
    <entity>
     <name>student</name>
     <field>ID</field>
    </entity>
   </node>
   <edge>
    <ID>EID001</ID>
    <predicate>works_in</predicate>
    <fromNodeID>NID0002</fromNodeID>
    <toNodeID>NID0001</toNodeID>
   </edge>
   <edge>
    <ID>EID002</ID>
    <predicate>majors_in</predicate>
    <fromNodeID>NID0003</fromNodeID>
```

```
<<continue>>
```

```
    <toNodeID>NID0001</toNodeID>
   </edge>
  </content>
 </defineGraphlog>
 <showGraphlog>
  <include>dept_people</include>
  <ID>showGraphlogq15</ID>
  <distinguished–show>
   <node>
    <ID>NID0001</ID>
    <entity>
     <name>student</name>
     <field>ID</field>
    </entity>
   </node>
   <node>
    <ID>NID0002</ID>
    <entity>
     <name>staff</name>
     <field>SID</field>
    </entity>
   </node>
  </distinguished–show>
  <content>
   <node>
    <ID>NID0003</ID>
    <entity>
     <name>dept</name>
     <field>No</field>
    </entity>
   </node>
   <node>
    <ID>NID0004</ID>
    <entity>
     <name>"Computing Science"</name>
    </entity>
   </node>
   <edge>
    <ID>EID001</ID>
```

```
<<continue>>
```

```
    <predicate>has_Name</predicate>
    <fromNodeID>NID0003</fromNodeID>
    <toNodeID>NID0004</toNodeID>
   </edge>
   <blob>
    <ID>BID001</ID>
    <predicate>dept_people</predicate>
    <outerNodeID>NID0003</outerNodeID>
    <innerNodeID>NID0002</innerNodeID>
    <innerNodeID>NID0001</innerNodeID>
   </blob>
  </content>
 </showGraphlog>
</graphlog>
```

Figure 44: TGL Program - Return Staff members and students in the Computing Science Department

```
This is Coral Version 1.5.2
-------------------------------------------------------

     Welcome to CORAL.

     All commands MUST end with a period .
     Type help. to access help information.

-------------------------------------------------------

ready>>ready>>ready>>ID=4881177, SID=cs0001.
... next answer ? (y/n/all)[y]ID=4125785, SID=cs0001.
... next answer ? (y/n/all)[y]ID=4881177, SID=css001.
... next answer ? (y/n/all)[y]ID=css001, SID=cs0001.
... next answer ? (y/n/all)[y]ID=4881177, SID=cs0002.
... next answer ? (y/n/all)[y]ID=4125785, SID=cs0002.
... next answer ? (y/n/all)[y]ID=css001, SID=cs0002.
... next answer ? (y/n/all)[y]ID=4125785, SID=css001.
... next answer ? (y/n/all)[y]ID=css001, SID=css001.
... next answer ? (y/n/all)[y]ID=4881177, SID=4125785.
... next answer ? (y/n/all)[y]ID=4125785, SID=4125785.
... next answer ? (y/n/all)[y]ID=css001, SID=4125785.
... next answer ? (y/n/all)[y](Number of Answers = 12)
ready>>
```

Figure 45: CORAL Execution Results - Return Staff members and students in the Computing Science Department

4.4.4.5 TGL Program for Results

Figure 46 gives the translated results that would be returned to the upper layer system.

Each item of answers in the CORAL execution results is translated. And the variables and their values are associated with the original elements in the graph. In this particular example, the variables SID and ID that represent the key attributes of entities staff and student are linked to theirs nodes respectively and the corresponding attribute positions are indicated too. Notice that the repetition of the instances for two entities are not eliminated. This is necessary because although in this example no other variables are interested, it is usual that many variables are involved in a single answer and judgment for eliminating duplication can not be given arbitrarily. This is not bad either since when the Graphical User Interface system attempts to assign values to the blob contained nodes, it can remove duplications from its collection at ease.

4.5 Increment Three Process

This section focuses on how to integrate TGL logical expressions, negation and aggregation into the translation system and wraps up the whole development. It completes the analysis, design and implementation of the system and generates the final deliverables.

4.5.1 Increment Three — Analysis

The analysis for Increment Three concentrates on complex translation circumstances. Besides directly using predicates for labels, TGL logical expressions can be involved to make more concise and more expressive queries. Additionally, negation and aggregation are two extended fields for logic querying. This part provides detail descriptions for these complex query techniques.

```
<showGraphlogReturn ID ="showGraphlogq15">
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >cs0001</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >4881177</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >cs0001</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >4125785</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >css001</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >4881177</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >cs0001</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >css001</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >cs0002</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >4881177</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >cs0002</field>
```

<< continue>>

```
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >4125785</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >cs0002</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >css001</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >css001</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >4125785</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >css001</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >css001</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >4125785</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >4881177</field>
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >4125785</field>
```

<< continue>>

```
  </node>
 </result>
 <result>
  <node ID ="NID0002" >
   <field pos ="0" >4125785</field>
  </node>
  <node ID ="NID0001" >
   <field pos ="0" >css001</field>
  </node>
 </result>
</showGraphlogReturn>
```

Figure 46: TGL Program Corresponds to the CORAL Execution Results

Use case	processUsingPositiveTGLLogicExpressions
Scenario	In the TGL programs submitted, labels could be either predicates or positive TGL logical expressions. the the translation of the expressions that act as labels need to be combined into the final CORAL programs. Then the translation system executes the CORAL programs with the suppport of the CORAL system. In the end, the transformed results are returned to the upper layer system.
Pre-condition	Well-formed TGL logical expressions as labels. Coral system is ready.
Post-condition	translated results in TGL programs.
Exception	TGL logical expressions are illegal according to the TGL. Input/Output errors. CORAL system is not accessible.
Use case	processInvolvingNegation
Scenario	The labels in the query patterns that users draw are involving negation. So the corresponding universe of discourse involves labels' environment. The translation system has to consider this particular condition during the translation. Then it could consults generated CORAL programs on the CORAL system. At the end, it returns translated results to the upper layer system.
Pre-condition	Well-formed TGL logical expressions that involve negations. The CORAL system is ready.
Post-condition	Results in the format of TGL program that corresponds to the CORAL execution results.
Exception	Negations are not properly defined. I/O errors.

Use case	processUtilizingAggregation
Scenario	In the TGL programs that define new relations, aggregation is conjuncted with the node labels or predicate attributes. The translation system should integrate the aggregation functions into the whole translation. Then with the generated CORAL programs, it fetch results from the CORAL system. However, the execution results received from the coal system have to be translated into TGL programs acceptable by the upper layer system.
Pre-condition	Legal aggregation functions applied to labels. TGL programs are in good conditions.
Post-condition	Well-defined CORAL programs that correspond to the definition.
Exception	The application of aggregation is not appropriate.

4.5.2 Increment Three — Design

Figure 47 illustrates the essential decomposition of the translation system. Thereby, system components are recognized and the innerconnections among them are identified.

4.5.2.1 Integrating the Translation of TGL logical Expressions

The translation of the graphlogs essentially includes translating the context of the query patterns and translating each element in the patterns. An element expressed using TGL logical expressions complicates the translation. This section focuses on the translation of these elements and the integration of the translation into those accomplished in Increment One and Increment Two.

Translation of TGL logical Expressions Basically, the translation of TGL logical Expressions shares the rule of that of Graphlog path expressions, and the Interpretations of the semantics are similar.

As regulated in the Transferable Graph Language, an logical expression follows the grammar: $E \leftarrow E \mid E; E \cdot E; -E; \neg E; (E); E+; E*; predicate(attributeList)$. The interpretation of such an expression is fundamentally restating the query objectives

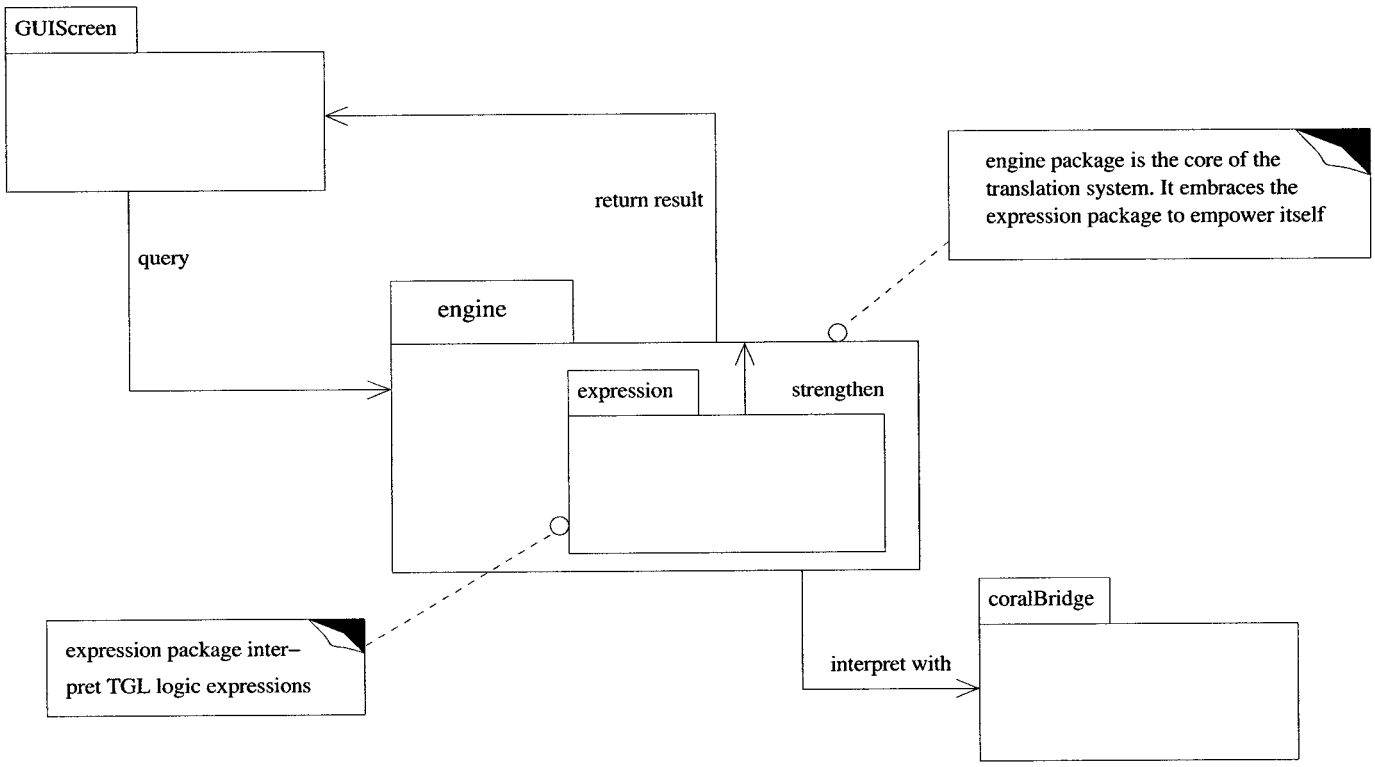


Figure 47: Final Conceptual Structure of the Translation System

in the rule head and describing subexpressions and their relationship in the rule body. The translation follows the steps: scanning an expression, breaking down the expression to tokens, recognizing the path regular pattern, setting up run-time data structure by organizing the token information, and last, generating expected code using the run-time data.

The collaboration of classes that serve together to translate TGL logical expressions is illustrated in Figure 48.

Token States Again the *State Design Pattern* is applied here. The TOKENIZER class relies on the token states to determine the type and value of a token, and there are five token states involved in the process.

1. PredicateState

   The PredicateState determine what comprise a predicate. The characters that are allowed in a predicate are: 'a' - 'z', 'A' - 'Z', '0' - '9', and '_'. By convention of logic programming language, a predicate begins with a lower letter, and numeric digits are allowable in a predicate. After a PREDICATESTATE object finishes a

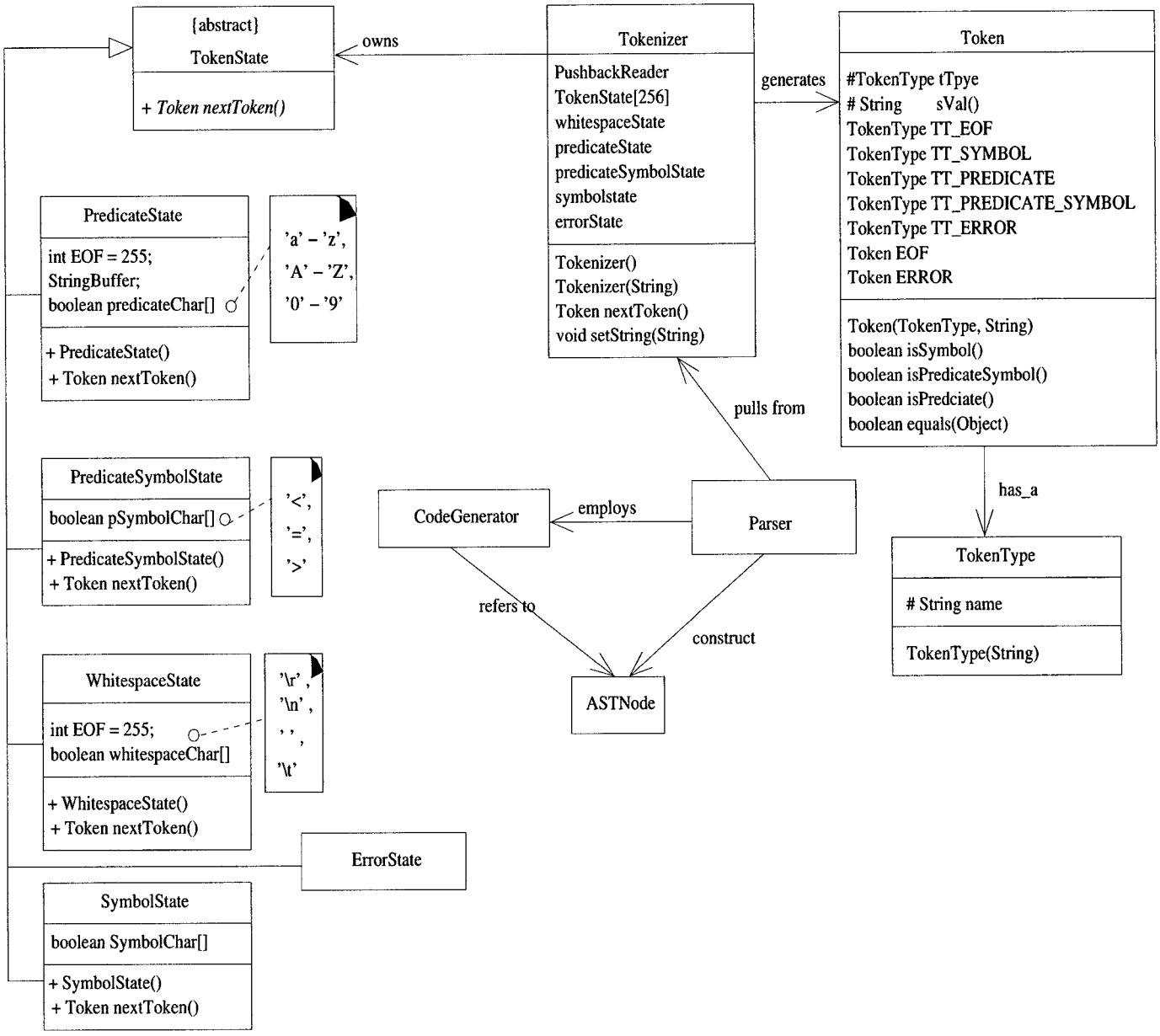


Figure 48: Translation TGL logical Expression Class Diagram

token, it transfers control back to the TOKENIZER object. PREDICATESTATE class utilizes a set of overloaded private methods with name of *setPredicateState* to internally legitimate these characters.

2. PredicateSymbolState
   The characters, ' <', ' >', *and* ' =' are admitted by the PredicateSymbolState. When the TOKENIZER object encounters one of them, it passes the character as well as itself to its internal PREDICATESYMBOLSTATE object. Then the PREDICATESYMBOLSTATE object continues to look up in the input stream to check whether a legal combination exists instead of just a single symbol. If a combination is matched, the PREDICATESYMBOLSTATE will return a token that takes the combination as its value. However, if the token contains only the character passed to the PREDICATESYMBOLSTATE object, the PREDICATESYMBOLSTATE object has to return back the character looked over to the input stream. This is accomplished through a PUSHBACKREADER object.

3. SymbolState
   A SymbolState recognizes the character set {'|', '.', '−', '¬', '*', '+', '(', ')' } . These characters are the conjunction operators of the TGL logical expressions. After the TOKENIZER object encounter one of the symbols, it decides the SYMBOLSTATE is responsible for the construction of next token, so it passes the symbol to its SYMBOLSTATE object.

4. WhiteSpaceState. The whitespace referred here is a sequence of one or more whitespace characters. The WHITESPACESTATE object collects as many whitespace characters as it can at one time and discards them, then it passes back the control to the Tokenizer object.

5. ErrorState
   If a character outside all the characters recognized by above token states appears, the TGL logical expression runs into error. The ERRORSTATE is responsible for dealing with this kind of problem and it informs the parser that an error exists.

TOKENIZER Class Driven by the PREDICATEPARSER, The TOKENIZER class relies on its token states to pull out tokens one by one. The TOKENIZER class could

set up its resource input stream when constructing. Another alternative is provided to promote the reuse of a TOKENIZER object: first constructing an TOKENIZER with or without specifying reading resource, and later using the method *setString(String)* to provide the resource.

PREDICATEPARSER **Class** The PREDICATEPARSER class is the engine of the whole translating TGL logical expression process. Figure 49 depicts the class design of PREDICATEPARSER. At the beginning, the client creates a PREDICATEPARSER object by sending it the expression to be analyzed and the information of associated nodes. Then the PREDICATEPARSER object immediately constructs its TOKENIZER object with the logical expression. After receiving the expression, the TOKENIZER object spits out TOKEN objects one after another until the end of the expression is reached through its token states objects. During the process, the PREDICATEPARSER is monitoring the TOKENIZER object and receiving every TOKEN object it generates. Along with the receiving, the PREDICATEPARSER object constructs an ASTNODE object that collects all the tokens in the expression. After the TOKENIZER object finishes scanning, the PREDICATEPARSER object completes the ASTNODE construction, too. Then the PREDICATEPARSER object calls a CODEGENERATOR object to work on the ASTNODE instance and produces a segment of CORAL program accordingly. The context-free grammar derived for recognizing the tokens is shown in Figure 50

ASTNODE **Class** ASTNode is the abbreviation of Abstract Syntax Tree Node. An ASTNode is constructed with Token objects by a PREDICATEPARSER OBJECT. And each ASTNode corresponds to an TGL logical expression. Essentially, every ASTNode has a root and two branches. The root is a string that represents the relation of the two branches. Each branch is just another ASTNode. There are three variations for ASTNode instances. The ASTNode constructions for the examples are listed in Figure 51

1. An ASTNode having two branches
   This kind of ASTNode is constructed for expressions having two operands plus an infix operator. Giving an example, the ASTNode for expression $E1|E2$.

2. An ASTNode having only left branch

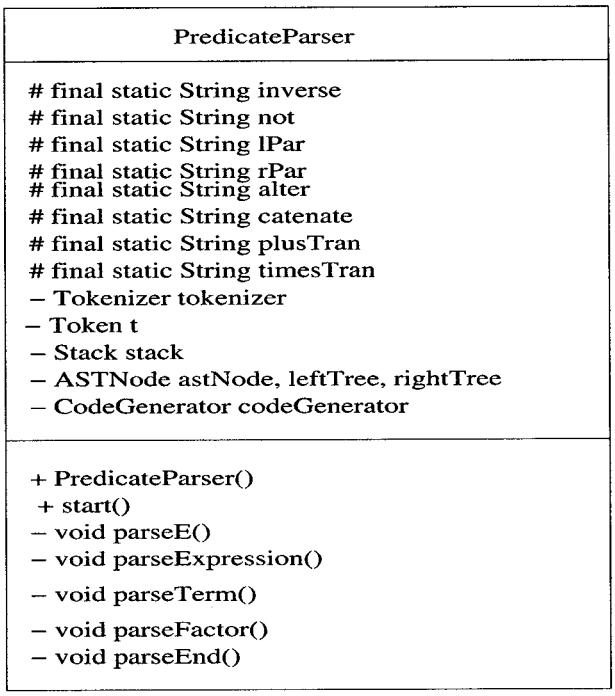


Figure 49: Class Diagram of PredicateParser

E := Expression | Expression
E := Expression.Expression
E := Expression
Expression := Term + Term
Expression := Term * Term
Term := -Factor
Factor := not End
Factor := End
Factor := (E)

Figure 50: The Context-free Grammar for TGL logical Expressions

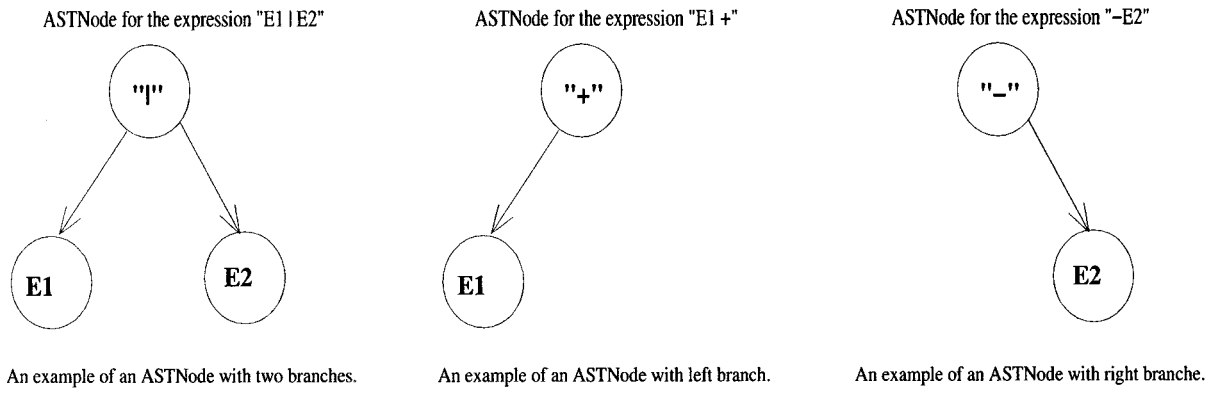


Figure 51: ASTNode Constructions Examples

This kind of ASTNode is constructed for expressions having one operand plus a suffix operator. An example is the ASTNode for expression $E1+$.

3. An ASTNode having only right branches
   This kind of ASTNode is constructed for expressions having one operand plus an prefix operator. An example is the ASTNode for expression $-E1$.

During the construction of an ASTNODE for a TGL logical expression, its left branch and right branch are updated continuously. New ASTNODE objects are added and old ones are updated. Thus, a deep copy, but not a shadow one, of an ASTNODE is required while transferring forward and backward. Class ASTNODE fulfills the function by inheriting the interface PUBLICLYCLONABLE. Figure 52 depicts the class ASTNode.

Producing CORAL Program for TGL logical Expression Class CODEGENERATOR is responsible for producing the CORAL program finally for a TGL logical Expression. However, it has to work on the intermediate ASTNODE generated by a PREDICATEPARSER object. The design of CODEGENERATOR is shown in Figure 53

Combining with Context Translation The translation of TGL logical expression serves as a part of the whole translation. It has to be integrated with the translation of context accomplished in Increment One and Increment Two. Since the TGL logical expressions serve as edge or blob labels, the translation of the expressions should integrate with the translation of the edge or blob. So the job is assigned to

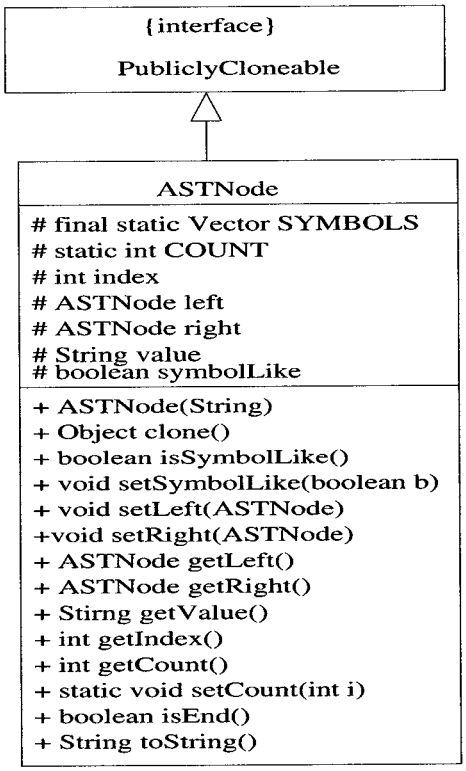


Figure 52: Design of Class ASTNode

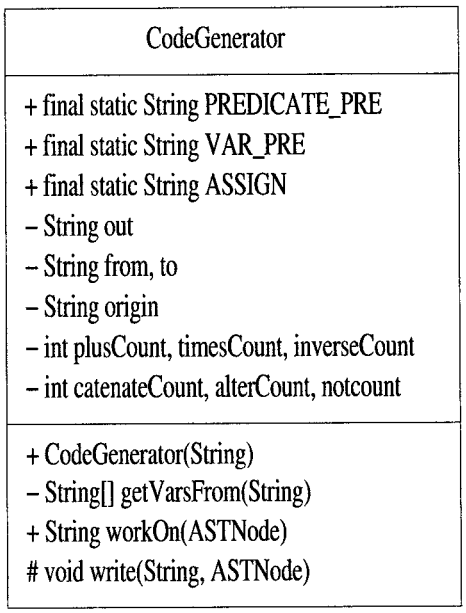


Figure 53: Design of Class CodeGenerator

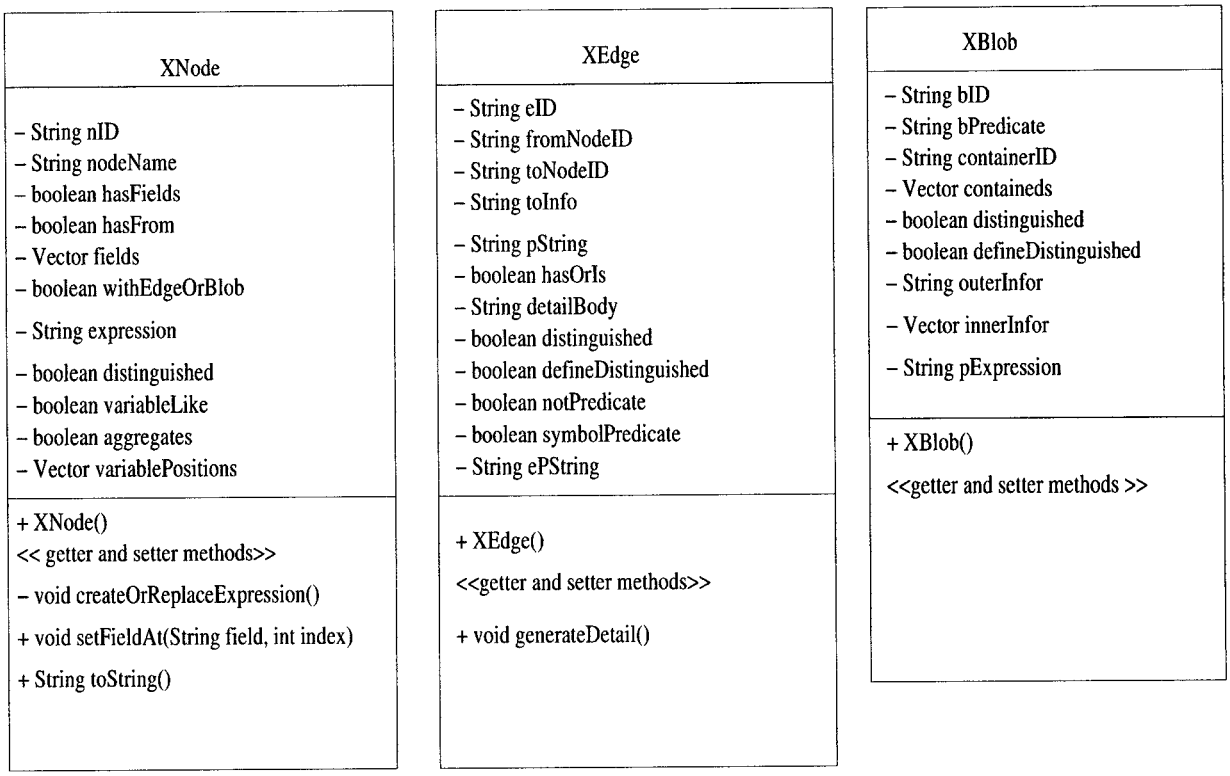


Figure 54: Design of classes XNode, XEdge and XBlob

the corresponding edge or blob. The *generateDetail* method will take care of this part of translation. When the XCODEGENERATOR need the definition of the edge or the blob, it simply origins the edge or blob itself to generate the translation of the TGL logical expression and retrieves the CORAL program segment.

4.5.2.2 Translation Involving Aggregation

Aggregate functions can only be used as arguments to define new relations. So if the label of a node is expressed involving aggregate functions, the translation segment of the node will not appear as a part of a rule body in the CORAL program. XNODE class is assigned to prevent the illegal listing. Its *isAggregate* method lets the CODEGENERATOR object check whether the label of a node contains an aggregate function. If it is true, the node is excluded from the constitution of the rule body.

Finally, the XNODE, XEDGE and XBLOB classes are complete. Their class designs are illustrated in Figure 54

4.5.3 Increment Three — Implementation

4.5.3.1 Translating TGL logical Expression Label for An Edge

The implementation of the method *generateDetail* for an edge is listed in Figure 55.

4.5.3.2 Translation Involving Negation

The negation involvement appears in two kinds of circumstances. First, if the predicate that corresponds to the TGL logical expressions is negative. Second, if part of logical expressions are negative. Dealing with these two circumstances need different strategies to deal with.

Labels Involving Negation If the TGL logical expression for a label uses the negation operator $\neg$, the translation of the negative part has to be incorporated into the translation of its level. It is not acceptable to use a temporary predicate representing the negative part to attend the translation of its level any more.

CODEGENERATOR class is responsible for the proper combination of negative parts with their level translation. It declares a private boolean method *isNotBranch* to check if the current ASTNODE branch being translated is negative. Upon a negative branch, the CODEGENERATOR object will add the translation of the branch directly to those of the branch's siblings.

Negative Labels In cases when not only the labels are constructed using $\neg$, but the final predicates that derived from the labels are negative, the outside context of the elements that have the labels is the universe of discourse for the negation.

In such cases, the translation of the labels should be listed directly with the translation of its context instead of just combining its temporary predicate representative with the context, and defining the negation translation before. Otherwise, it will cause execution errors and no expected results will be derived. Besides, all the variables that serve to the negation should be bound before the rule, which involves the negation, is evaluated.

To attain this goal, the situation should be identified before generating the highest level translation. The situation is judged through the *isNotPredicate* method of the elements. Once the situation is identified, it should be separated from the normal ones so that later it is dealt with individually. In XDEFINECODEGENERATOR and

```
public void generateDetail(){
 // only serve for hasOrIs field is false
 try{
       if(hasOrIs == false){
             PredicateParser predicateParser = new PredicateParser(pString,fromInfo,toInfo, eID);

             detailBody = predicateParser.start();
             notPredicate = predicateParser.rootIsNot();
             symbolPredicate = predicateParser.isSymbolPredicate();

             if(notPredicate){ // not ...
              // note the first goal is at the end of detailBody.
               int firstGoal = detailBody.lastIndexOf(".");

               detailBody = detailBody.substring(0, firstGoal);

               firstGoal = detailBody.lastIndexOf(".");

               ePString = detailBody.substring(firstGoal+1);

               detailBody = detailBody.substring(0, firstGoal+1);

               int firstAssign = ePString.indexOf(":-");
               ePString = ePString.substring(firstAssign+2).trim();

             }else if(symbolPredicate){
              int assignPos = detailBody.indexOf(":-");
              int goalEnd = detailBody.lastIndexOf(".");
              ePString = detailBody.substring(assignPos+2, goalEnd);
              System.out.println("L108 in XEdge, ePString: " + ePString);
             }else{ // normal predicate
               int pos = pString.indexOf('(');
               if(pos>0){ //predicate(otherSuffix).
                 ePString = eID.trim().toLowerCase() + "(" + fromInfo + "," + toInfo + "," +
            pString.substring(pos+1);
               }else{ // just predicate
                 ePString = eID.trim().toLowerCase() + "(" + fromInfo + "," + toInfo + ")";
               }
             }
       }
 }catch(IOException e){ System.out.println("IO errors in class XEdge" + e) ;}
}
```

Figure 55: Implementation of Detail of An Edge

XSHOWCODEGENERATOR classes, a vector pNotString is used to record the appropriate CORAL program segments that corresponds to the negative labels. There are two benefits using this pNotString vector. First, the elements involing negation can be identified and integrated correctly. Second, it allows for evaluation of positive predicates in advance so that variables for the negative predicates may bind values before the evaluation.

4.5.4 Increment Three — Test

The requirements in Increment Three emphasize on utilizing TGL logical expressions and involving negation and aggregation. The analysis, design and implementation deliverables in turn are centered on the translation of the logical expressions, negation, and aggregation and the integration of these parts to those accomplished in Increment One and Increment Two.

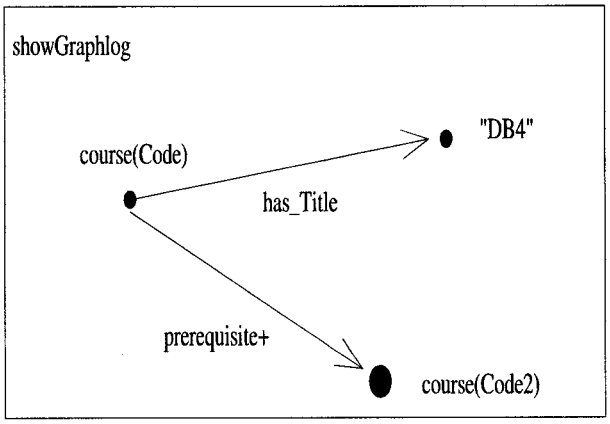
Because the functionality and capabilities of positive TGL logical expressions, negation, and aggregation vary significantly, test cases are designed and carried on respectively. In each test case, first, a possible graph to express the query example is given. Next, the corresponding TGL program is provided for the graph. Then, generated CORAL programs for the TGL program are listed. And finally, the TGL program that corresponds to the CORAL execution results are presented.

4.5.4.1 Positive TGL logical Expressions – Return Direct and Indirect Prerequisite Courses for the "DB4" Course

Graph&TGL Program This is a typical recursive query. Not only the direct prerequisite courses of the course "DB4" are interested, but those of course "DB4"'s direct prerequisite courses. In the next turn, more indirect courses will be discovered. This search will carry on until no more new courses can be found.

The recursive queries are obviously beyond the ability of SQL. But under the support of our TGL and translation system, the query can be expressed and processed elegantly. A possible graph query expression and the corresponding TGL program could be:

This query can be denoted using the TGL logical expression *prerequisite+*, which is composed of by the system schema *prerequisite* and the suffix TGL logical operator




```
-- Return all direct and indirect prerequisite courses for
-- the "DB4" course.

<graphlog>
 <showGraphLog>
  <ID>showGraphlogq18</ID>
  <distinguished-show>
    <node>
     <ID>NID0003</ID>
     <entity>
       <name>course</name>
       <field>Code2</field>
     </entity>
    </node>
  </distinguished-show>
  <content>
   <node>
    <ID>NID0001</ID>
    <entity>
      <name>course</name>
      <field>Code1</field>
    </entity>
   </node>
   <node>
    <ID>NID0002</ID>
    <entity>
      <name>"DB4"</name>
    </entity>
   </node>

  -- continue here

   <edge>
    <ID> EID001</ID>
    <predicate>has_Title</predicate>
    <FromNodeID>NID0001</FromNodeID>
    <ToNodeID>NID0002</ToNodeID>
   </edge>
   <edge>
    <ID> EID002</ID>
    <predicate>prerequisites+</predicate>
    <FromNodeID>NID0001</FromNodeID>
    <ToNodeID>NID0003</ToNodeID>
   </edge>
  </content>
 </showGraphLog>
</graphlog>
```

Figure 56: Graph&TGL Program for an Query Example Using Positive TGL logical Expressions

```
module showGraphlogq18.
export showGraphlogq18(f).

pPlusCount1(Code1,Code2) :- prerequisites(Code1,Code2).
eid002(Code1,Code2) :- pPlusCount1(Code1,VAR_Plus1),eid002(VAR_Plus1,Code2).
eid002(Code1,Code2) :- pPlusCount1(Code1,Code2).
showGraphlogq18(Code2):-
course(Code1,"DB4",_),
eid002(Code1,Code2).
end_module.

?showGraphlogq18(Code2).
```

Table 11: The Coral Program That Query Using a TGL logical Expression

+ . The semi-built-in predicate *has_Title* introduces the related course with the title of "DB4".

The corresponding TGL program is listed under the graph. The distinguished node with label "course(Code2)" is enclosed by the open tag and closing tag of *distinguished-show*. Other context information are described in the part of content. It is clear that this query pattern is simple but powerful.

Generated CORAL Program The translation system parses this program and generates an executable CORAL program(in Table 11)for later use.

TGL Program for Results Through executing the CORAL program in Table 11, the CORAL system return the answers we expected. Then, the translation system generates the corresponding TGL program shown in Table 12.

4.5.4.2 Involving Negation – Return Students Taking Only Courses Given by Steve Johnson

Graph This is a complex query, which can not be expressed without the support of negation. There are students that take courses taught by staff member Steve Johnson. They could be divided into two categories: students who take courses not only from Steve Johnson but also from other staff members, and those who do not takes courses given by staff members other than Steve Johnson. It is not difficult to find the whole

$< showGraphlogReturnID =$ "$showGraphlogq18$" >
$< result >$
$< nodeID =$ "$NID0003$" >
$< fieldpos =$ "0" > $comp646 < /field >$
$< /node >$
$< /result >$
$< result >$
$< nodeID =$ "$NID0003$" >
$< fieldpos =$ "0" > $comp248 < /field >$
$< /node >$
$< /result >$
$< result >$
$< nodeID =$ "$NID0003$" >
$< fieldpos =$ "0" > $comp218 < /field >$
$< /node >$
$< /result >$
$< /showGraphlogReturn >$

Table 12: The Result TGL Program For a Query Using a TGL logical Expression

set. The puzzling task is to separate the first part of students from the whole set. Negation is used to exclude the unnecessary part from the whole range.

Figure 57 provides a solution. Note that the character reference "&#00AC;" is used to represent ¬ in the picture because ¬ can not be input from keyboard. In fact, Graphical User Interface system could provide a button with ¬ as label to users for the consistency of representation and transfer to TGL program using the "&#00AC;" behind the scene.

The three query patterns represent three steps to process the query. The two define query patterns support the final show query pattern. First, a relation students_by_staff is declared to collect all students that take courses given by a specific staff member. The second definition, students_not_SJ, is based on the first definition and underlying system schemes. It is used to search all students that take courses from staff members whose names are not Steve Johnson. A key, or tricky, thinking here is that besides the course from other staff members, a student in this set still may take courses from Steve Johnson. Last, the desirable answers could be obtained through the show query. It discards all students that take courses from both Steve Johnson and other staff members from the whole set of students who attend courses

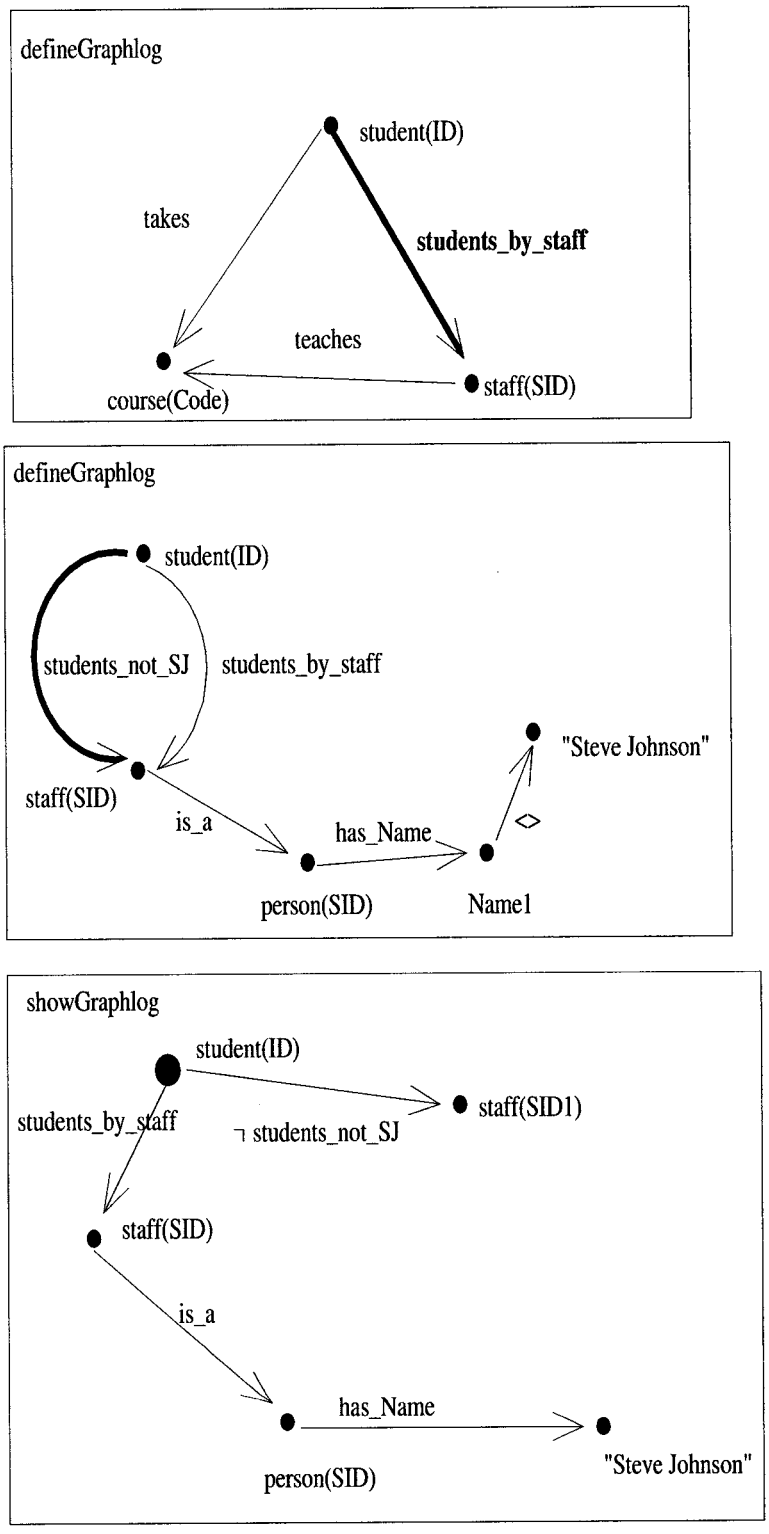


Figure 57: Graph for an Query Example Involving Negation

```
module showGraphlogq12.
export showGraphlogq12(f).

pNotCount1(ID,SID1) :- students_not_SJ(ID,SID1).
eid001(ID,SID) :- students_by_staff(ID,SID).
showGraphlogq12(ID):-
person(SID,"Steve Johnson"),
eid001(ID,SID),
not pNotCount1(ID,SID1).
end_module.

?showGraphlogq12(ID).
```

Table 13: The Coral Program That Query Involving Negation

from Steve Johnson.

Corresponding TGL Programs Only the TGL program for the final show query pattern provides an idea of how negation is translated and integrated.

TGL Program for Results Through executing the CORAL program in Table 13, the CORAL system return the answers we expected. Then, the results are translated into an acceptable TGL program as shown in Table 14.

The results could be judged based on the facts in file uni.F.

4.5.4.3 Involving Aggregation – Return Courses with Less Than Two Assessments

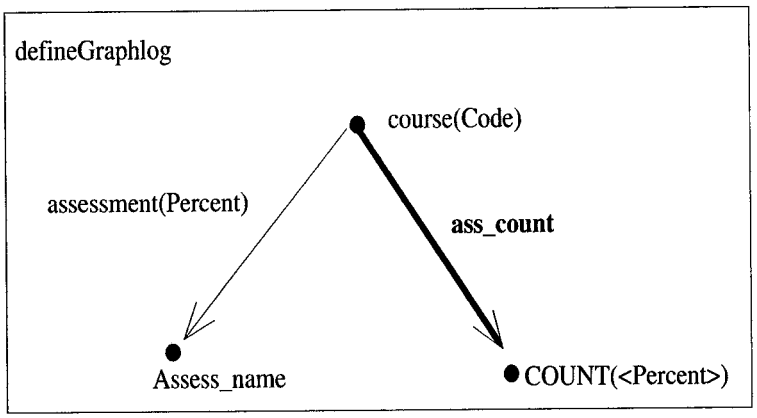
Graph Since counting is needed, this query has to employ aggregation functions. A possible solution is depicted in Figure 58.

Note that the aggregation function COUNT is used in the definition of relation ass_count in the first picture of the figure. In fact the to-node connects to the edge with label ass_count has $COUNT(< Percent >)$ as its label. Later the newly defined relation is the base of the final query.

Corresponding TGL Programs The TGL program that corresponds to the definition of ass_count is shown in Table 15.

```
< showGraphlogReturnID = "showGraphlogq12" >
< result >
< nodeID = "NID0001" >
< fieldpos = "0" > 3345167 < /field >
< /node >
< /result >
< result >
< nodeID = "NID0001" >
< fieldpos = "0" > 3788947 < /field >
< /node >
< /result >
< /showGraphlogReturn >
```

Table 14: The TGL Program for Results of the Query Involving Negation



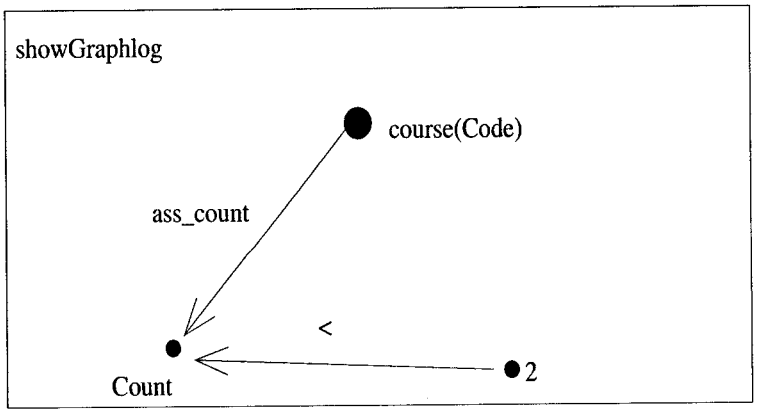


Figure 58: Graph for an Query Example Involving Aggregation

```
module ass_count.
export ass_count(ff).
eid002(Code,Assess_name,Percent) :-
assessment(Code,Assess_name,Percent).
ass_count(Code, count(< Percent >)) : −
eid002(Code,Assess_name,Percent).
end_module.
```

Table 15: The Coral Program That Query Involving Aggregation

```
< showGraphlogReturnID = "showGraphlogq21" >
< result >
< nodeID = "NID0001" >
< fieldpos = "0" > comp248 < /field >
< /node >
< /result >
< /showGraphlogReturn >
```

Table 16: The TGL Program for Results of the Query Involving Aggregation

TGL Program for Results Based on the newly defined relation using the CORAL program shown in Table 16, the final query results received from the CORAL system is translated as follows:

In effect, only one course that has the key of "comp248" meet the requirements.

Chapter 5

Conclusion

This thesis is rooted in the research requirements of biological scientists. First, the biological information is potentially very large and the structures of the information are significantly complex. So efficiency in organizing and processing these data is crucial for biological science. The CORAL system provides efficient support in the following two aspects:

1. CORAL allows users to create new abstract data types and integrate them with its query language. Thus, high-level types and corresponding methods such as matching and indexing for biological data can be defined. It avoids the inefficiency of using low level data types like list to simulating the biological data.

2. CORAL possesses the features of efficient treatment of large relations, aggregate functions, declarative semantics, powerful inference capabilities, and support of incomplete and structured data.

Second, the biological scientists expect an user interface to be as much user-friendly as possible to analyze the biological information. So far, a graphical user interface is the best choice. Although CORAL is useful and expressive, users have to write its programs and use a command-line query interface, which is quite impractical for biological projects.

Therefore, we want to implement a diagrammatic query system to support biological projects. Graphlog and Hy+ system present a useful and practical idea for diagrammatic query system based on the CORAL system. However, the implementation of Hy+ system is in Smalltalk, which limits the extension and reusability of

the system.

Our project borrows ideas from Graphlog and Hy+ system to support diagrammatic queries. But the application will be implemented in the Java language for the purpose of having extensibility, reusability and portability.

The application has three main parts: GUI system, a translation system, and backend database support. This thesis designs and implements the translation system. Also it proposes a textual language(Transferable Graph Language) used to record and transfer the graph queries composed by the end user in the GUI system. The Transferable Graph Language also regulates the format for the results sending back to the GUI system.

The main contribution of the thesis is as follows:

- It supports recursive queries, complex terms, negation, aggregation, and complex logic expressions.
- It extends the Graphlog by allowing a family of built-in predicates. The "is_a" and "has_XXX" predicates are user-friendly.
- It defines the transferable textual language, Tranferable Graph Language, which regulates the restatement of the query composition diagrams. The language is used to collect the information in the diagrams that contains the query expressions. In effect, this language itself could be used to compose queries since it has well-formed programming formats that are friendly to human beings. The more important feature is that the programs written in the language can be transfered across distributed computer systems, thereby increase the reusability and performance.
- It designs and implements a platform-independent translation system that is the middle layer of our diagrammatic query system. Further, the realization process conforms to Object-Oriented technologies and the implementation language is the Java programming language. As a consequence, the translation system has high reusability and good maintainability so that it could be planted into an internet or intranet application to support a graph query environment without changes.
- It tests the translation system and the Transferable Graph Language based on a university data model because this domain is more familiar to most users. Thus

the explanation of the query expressions and results are more understandable, and readers will not balk because of the sophisticated domain knowledge. The queries are taken in whole from the thesis "Evaluating Object-Oriented Query languages" [10]. Originally, these queries are designed and evaluated against the key features and functions of object-oriented query languages. In effect, the construction of these queries requires not only all the functions of a typical relational database query language, but those specific to object-oriented query language. So the successful composition of these queries using the Transferable Graph Language also proves that TGL has sufficient features to act as a query language.

The thesis is limited in the following ways: First, the thesis tests the translation system and the Transferable Graph Language using university data model instead of bioinformatics data model. Second, the current implementation supports only single PC mode execution although it leaves the interface to support network collaboration. Third, the GUI system and back-end database support are not finished yet, so the interface between the translation system and the two layers may require further test and adjustment.

Therefore, in the future, we can make the following improvements: first, we could design a typical biological data model to test the translation system. Especially, we could make use of the optimization mechanism of the CORAL system to improve the performance and increase the efficiency. Second, later the flesh-out and customization of REMOTETRANSLATOR will support queries through internet. Third, we need to test the application as a whole to adjust the interface between them.

Bibliography

[1] Greg Butler, Erich Bornberg-Bauer, Gosta Grahne, Franz Kurfess, Clement Lam, Joey Paquet, Isabel Rojas, Rajjan Shinghal, Lixin Tao, Adrian Tsang, *The BioIT Projects: Internet, Database and Software Technology Applied to Bioinformatics*, International conference on advances in infrastructure for electronic business, science and education on the internet, 2000.

[2] Jean B. Rogers, *A Prolog primer*, Addison-Wesley, 1986.

[3] Mariano P. Consens, Alberto O.Mendelzon, and Dimitra Vista, *Deductive Database Support for Data Visualization*, Proceedings of the 4th International Conference on Extending Database Technology, pages 45 – 58, 1994.

[4] Mariano P. Consens, Frank Ch. Eigler, Masum Z. Hasan, *Architecture and Applications of the Hy+ Visualization System*, IBM Systems J.,33(3): 458 – 476, 1994.

[5] Mariano P. Consens, Frank Ch.Eigler, Sergio R. Faria, *Hy+ User's Manual*, University of Toronto, 1993.

[6] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, Praveen Seshadri, *The CORAL Deductive System*, The VLDB Journal, 3:161 – 210, 1994.

[7] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, *CORAL — Control, Relations and Logic*, Proceedings of the 18th VLDB Conference, pp. 238 – 250, 1992.

[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Welsley, 1997.

[9] Steven John Metsker, *Building Parsers with Java*, Addison Welsley, 2001.

[10] Daniel K.C. Chan, Philip W. Trinder, Raymond C. Welland, *Evaluating Object-Oriented Query Languages*, The Computer Journal, Vol.37, No.10, pp. 858 – 872, 1994.

[11] Shel Siegel, *Object-Oriented Software Testing — A Hierachical Approach*, John Wiley, 1996.

[12] Norman Desmarais, *The ABCs of XML — The Librarian's Guide to the eXtensible Markup Language*, New Technology Press, 2000.

[13] David A. Watt, Deryck F Brown, *Programming Language Processors in Java*, Prentice Hall PTR, 1999.

[14] Matanya Pitts, *XML in Record Time*, San Francisco : Sybex, 1999.

[15] McLaughlin Brett, *Java&XML*, Sebastopol, CA : Cambridge, 2001.

[16] Len Bass, Paul Clements, Rich Kazman, *Software Architecture in Practice*, Addison Welsley, 2001.

[17] Bruce Eckel, *Thinking in Java*, Prentice Hall, 2000.

[18] Dennis de Champeaux, Douglas Lea, Penelope Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.

[19] Jess Garms, Daniel Somerfield, *Professional Java Security*, Birmingham, UK : Wrox Press, 2001.

[20] Darren Govori, *Java Application Frameworks*, New York : Wiley, 1999.

[21] Kafura Dennis, *Object-Oriented Software Design and Construction with Java*, Prentice Hall, 2000.

[22] Rowlett Tom, *The Object-Oriented Development Process:Developing and managing a robust process for Object-Oriented Development*, Prentice Hall, 2001.

[23] Russel Winder, Graham Roberts, *Developing Java Software*, Worldwide Series in Computer Science, 2000.

[24] XiaoPing Jia, *Object-Oriented Software Development Using Java*, Addison Welsley, 2000.

Appendix A

University Data Model System Schemes and Example Facts

Appendix A shows the underlying system schemes for the university data model. It also includes the records of the tables(or facts) for the purpose of testing and processing example queries.

Table 17 presents the underlying system schemes.

Below is the list of the sample records used to retrieve query results and test.

```
person(cs0001, "William Atwood").
person(cs0002, "Esmail Nabil").
person(ce0001, "Bird Herry").
person(css001, "Burger Losa").
person(ce1005, "Lee Joey").
person(4881177, "Steve Johnson").
person(3345167, "Dini Sabin").
person(3511786, "Grogono Gosta").
person(3788947, "Lui Lixin").
person(4125785, "Li Joey").
person(cs0003, "Steve Johnson").
person(cs0004, "Bob Campbell").

staff(cs0001, 2500).
staff(cs0002, 1800).
```

person(ID, Name).
staff(ID,Salary).
student(ID).
tutor(ID).
visiting_staff(ID).
dept(No, Name).
course(Code, Title, Credit).
address(AID, Street, District, City).
works_in(ID, Dept).
teaches(ID, Course).
majors_in(ID, Dept).
takes(ID,Course).
run_by(Course,Dept).
prerequisites(Course,PreCourse).
assessment(Course,assName,Percent).
lives_in(ID, AID).
first_supervisor(StaffID, StudentID).
second_supervisor(StaffID, StudentID).
resides(Dept, AID).

Table 17: The Underlying System Scheme for the Univesity Data Model

```
staff(ce0001, 4801).
staff(css001, 3400).
staff(ce1005, 1685).
staff(4125785,3400).
staff(cs0003,2300).

student(4881177).
student(3345167).
student(3511786).
student(3788947).
student(4125785).
student(css001).

tutor(css001).
tutor(4125785).

visiting_staff(ce1005).
visiting_staff(cs0001).

dept(cs, "Computing Science").
dept(gm, "John molson business").
dept(artG, "Art Gallery").
dept(edu, "Education").
dept(eg, "Engineering").

course(comp218, "Fundamentals of C++ Programming", 3).
course(comp248, "Introduction to Programming",3).
course(coen60, "Software Regular ",4).
course(comp651, "DB4", 4).
course(comp646, "Computer Networks and Protocols", 4).
course(eSL207, "English as Second Language 207", 1).

address(addr001, "Maisonuvue Street", "center ville", "Montreal").
```

```
address(addr002, "Hillhead Street", "Hillhead", "Montreal").
address(addr003, "University Avenue", "Kelvinside", "Montreal").
address(addr004, "Lincoln Street", "Dowanhill", "Glasgow").

works_in(cs0001,cs).
works_in(cs0002, cs).
works_in(ce0001,eg).
works_in(css001, cs).
works_in(ce1005, eg).
works_in(4125785, gm).

teaches(cs0001,comp676).
teaches(cs0002,comp218).
teaches(ce0001, coen60).
teaches(css001, comp646).
teaches(ce1005, coen61).
teaches(cs0003, comp248).
teaches(ce0001, eSL207).

majors_in(4881177,cs).
majors_in(css001,cs).
majors_in(3345167,gm).
majors_in(3511786,eg).
majors_in(3788947,edu).
majors_in(4125785,cs).

supervises(cs0001, 4881177).
supervises(css001, 4125785).
supervises(css001, 4881177).
supervises(cs0003, 3788947).
supervises(cs0004, 3788947).
supervises(cs0003, 3345167).
```

```
takes(4881177, comp646).
takes(4125785, comp218).
takes(4881177, eSL207).
takes(4125785, eSL207).
takes(4881177, comp248).
takes(4125785, comp248).%for qs1.P to be commented.
takes(3345167, comp248).
takes(3788947, comp248).

run_by(comp646, cs).
run_by(comp218,cs).
run_by(comp248,cs).
run_by(eSL207,edu).

prerequisites(comp248, comp218).
prerequisites(comp646, comp248).
prerequisites(comp651, comp646).

assessment(comp646,"mid-term1", 0.25).
assessment(comp646,"mid-term2",0.25).
assessment(comp646,"assign",0.25).
assessment(comp646,"final", 0.25).
assessment(comp651,"mid-term",0.25).
assessment(comp651,"project",0.25).
assessment(comp651,"final", 0.5).
assessment(coen60,"ass1", 0.33).
assessment(coen60,"ass2", 0.34).
assessment(coen60,"final", 0.33).
assessment(comp218,"mid-term", 0.5).
assessment(comp218,"final", 0.5).
assessment(comp248, "final", 1).
assessment(eSL207, "ass1", 0.1).
assessment(eSL207,"ass2", 0.1).
```

```
assessment(eSL207, "ass3", 0.1).
assessment(eSL207,"ass4", 0.1).
assessment(eSL207,"final", 0.6).

lives_in(cs0001, addr002).
lives_in(cs0002, addr003).
lives_in(4881177, addr001).
lives_in(4125785, addr004).
lives_in(3511786, addr002).

first_supervisor(cs0001, 4881177).
first_supervisor(cs0002,3511786).
first_supervisor(cs0003, 3788947).
first_supervisor(cs0003,4125785).

second_supervisor(cs0004,4125785).
second_supervisor(css001, 4881177).
second_supervisor(cs0004, 3788947).

resides(cs,addr002).
resides(gm,addr003).
resides(artG,addr001).
resides(edu,addr004).
resides(eg,addr003).
```

Appendix B

Sample Query Diagram Templates

This appendix presents diagram solutions for example queries in the university data model. It provides the corresponding TGL programs for these diagram queries as well. Finally, the TGL programs generated by translating the coral execution results of those queries are presented.

1. Query Diagrams
   Note that Q12, Q15, Q21 are not in the examples because they have been explained in detail in the test cases of Increment Two and Increment Three.

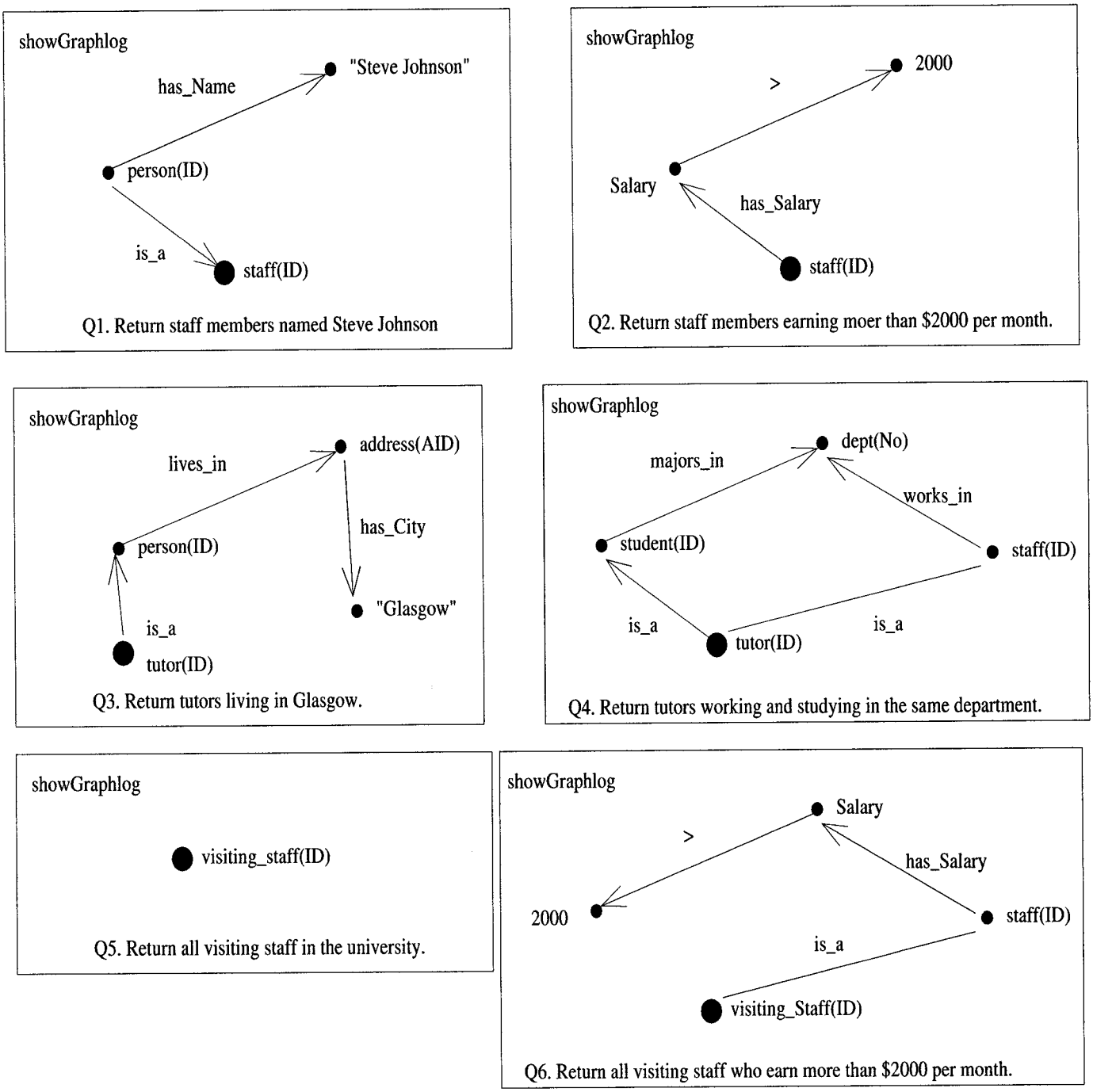
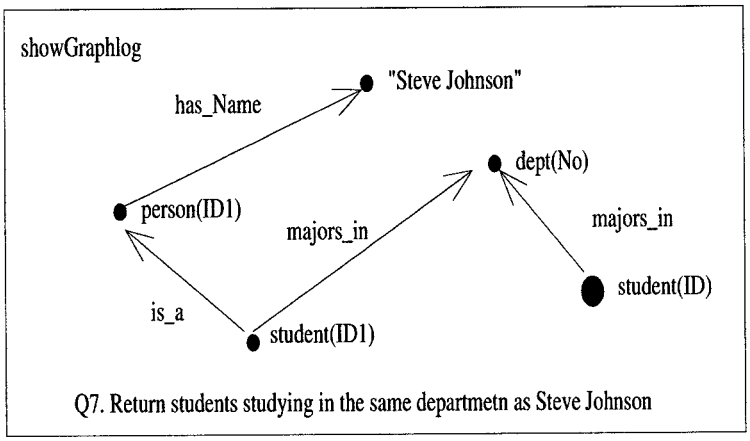
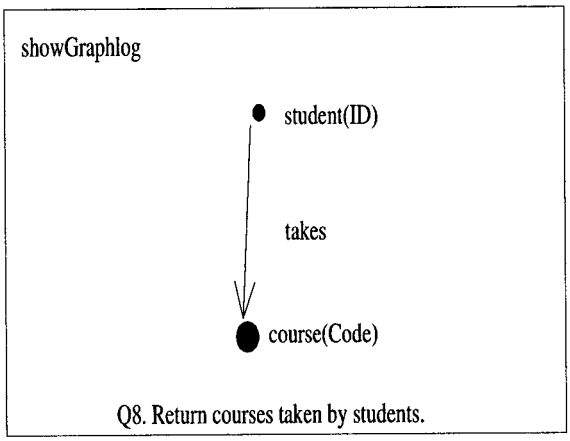
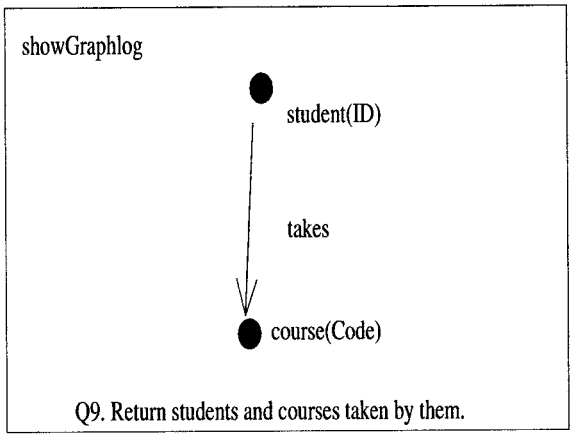
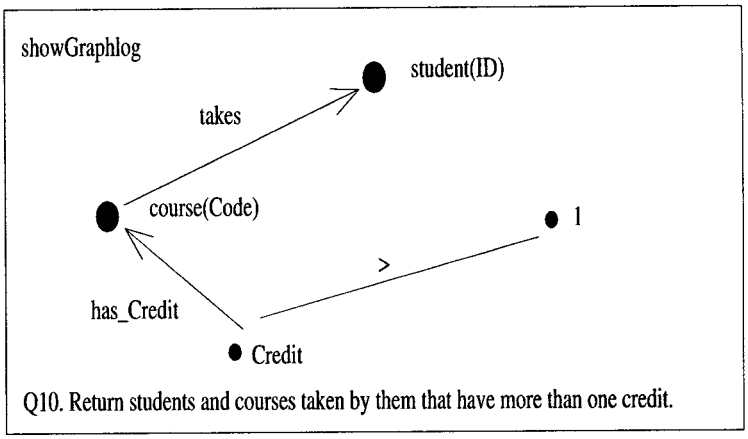


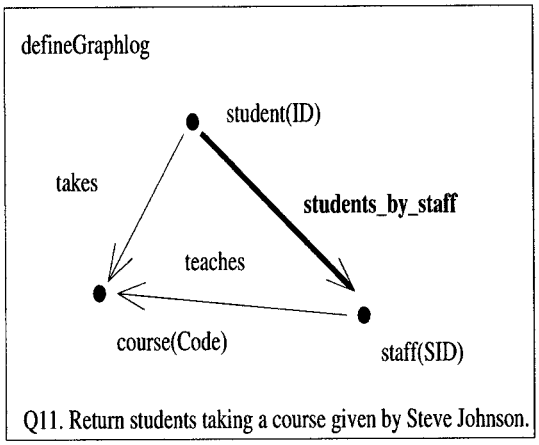
Figure 59: Diagrams of Example Query Composition(1 – 6)











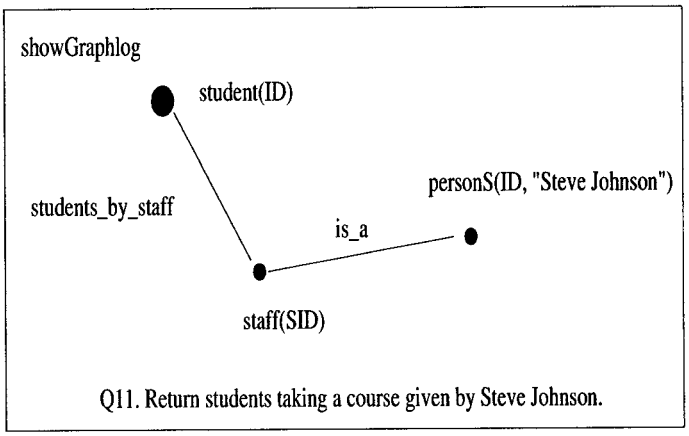
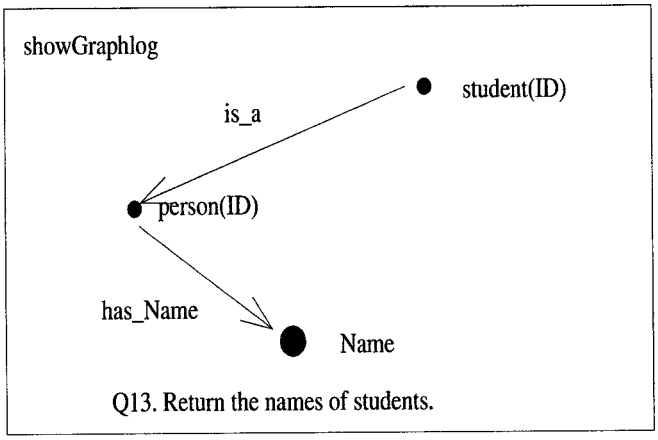
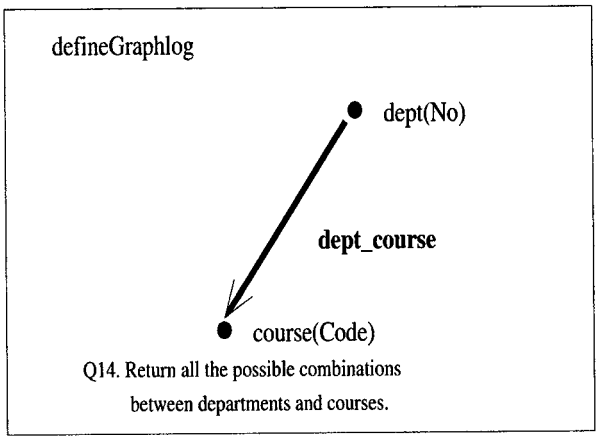
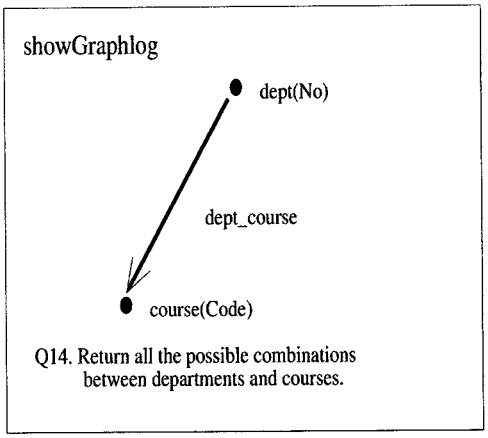
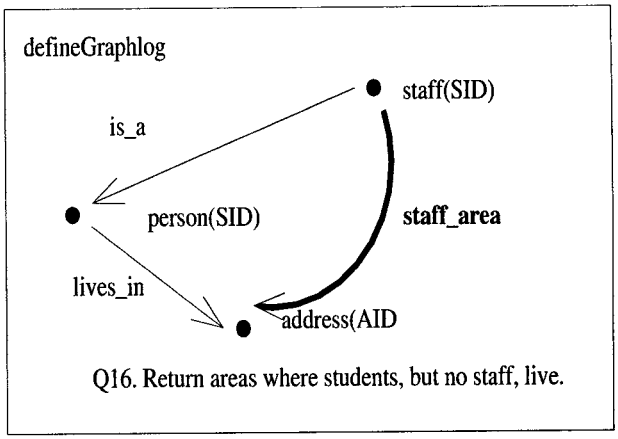


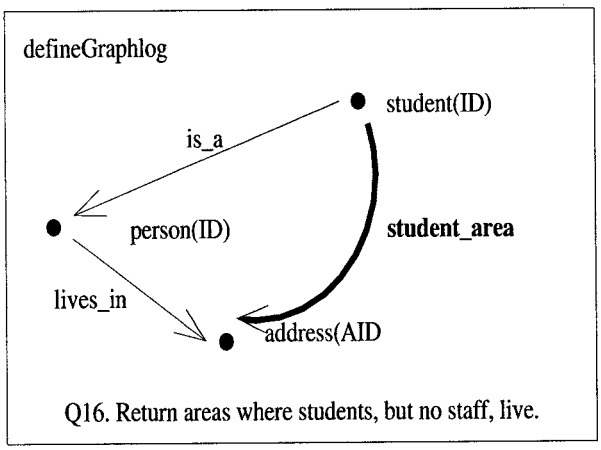
Figure 60: Diagrams of Example Query Composition(7 – 11)











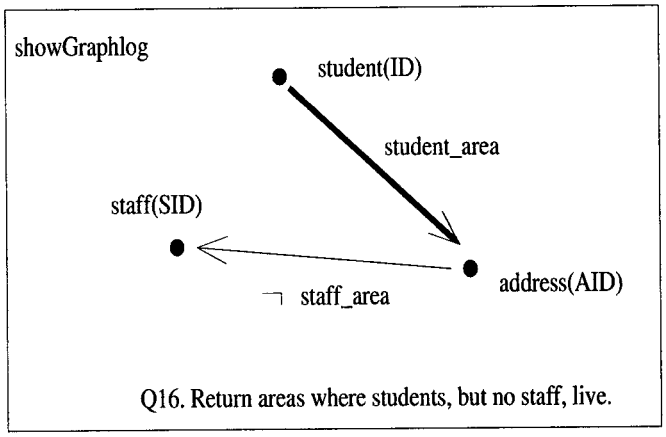
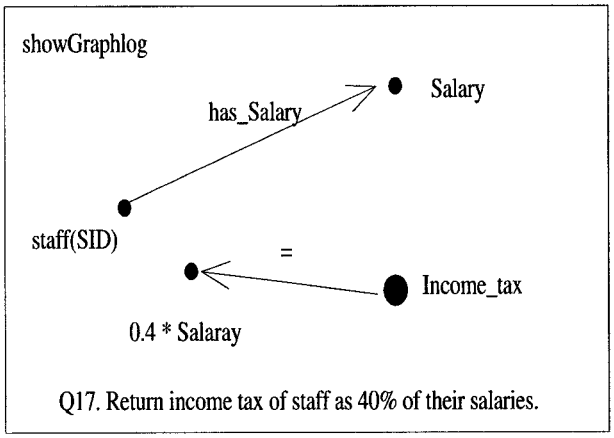
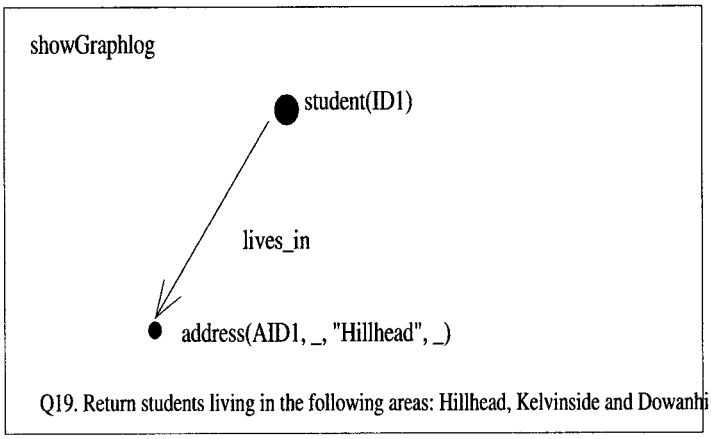
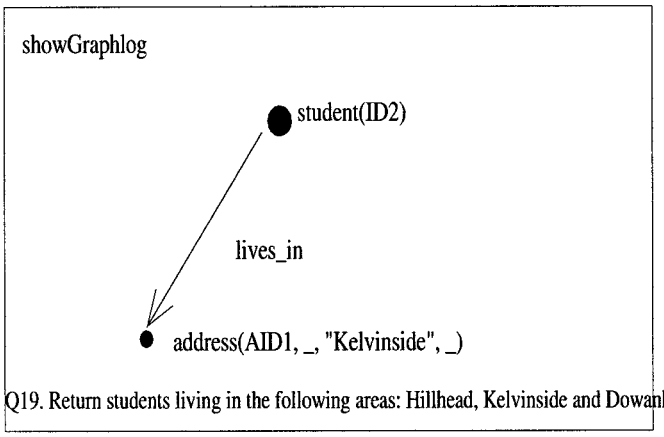
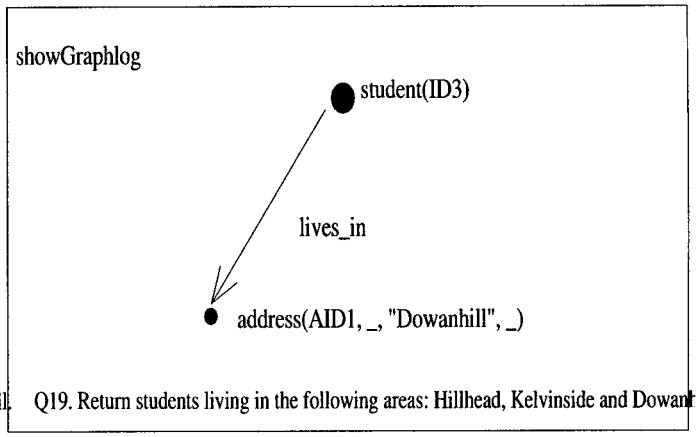


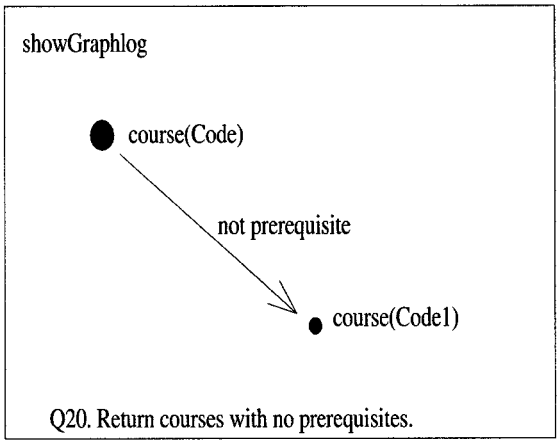
Figure 61: Diagrams of Example Query Composition(13 – 16)











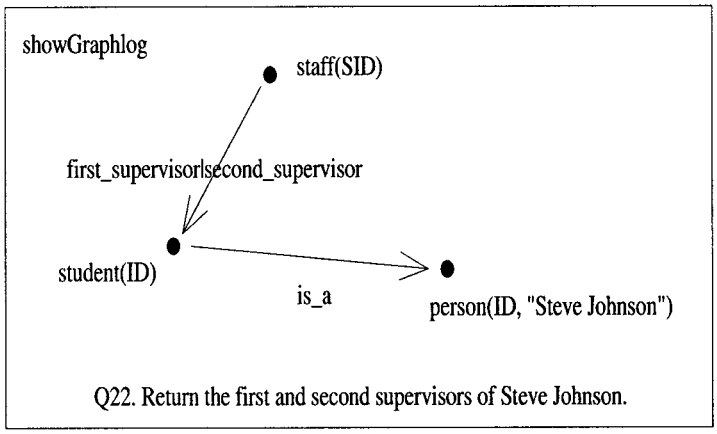
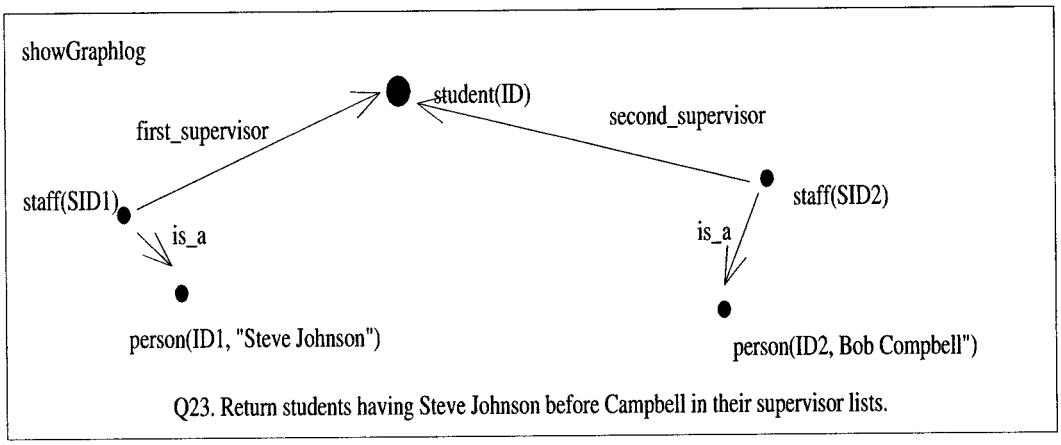
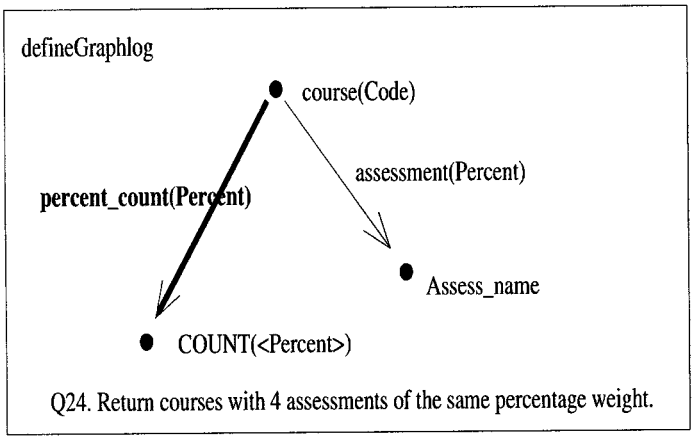
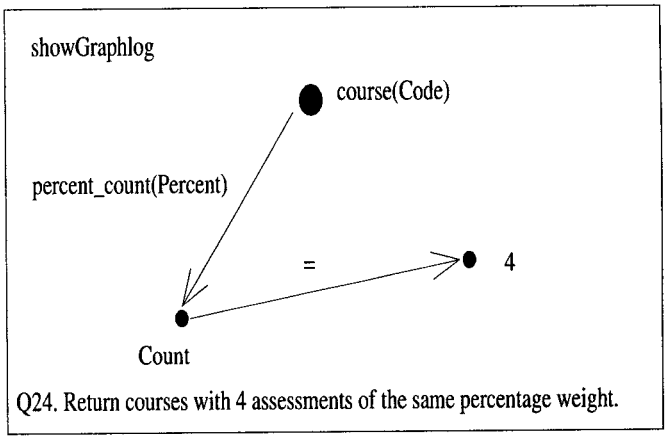
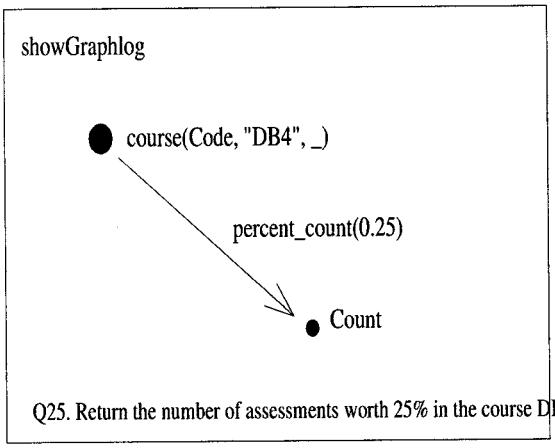


Figure 62: Diagrams of Example Query Composition(17 – 22)









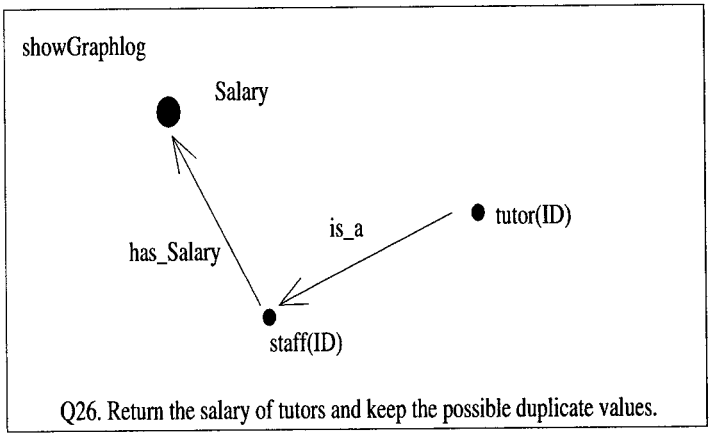
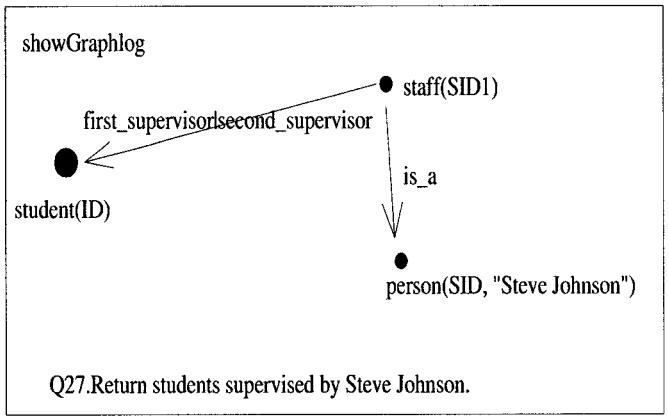
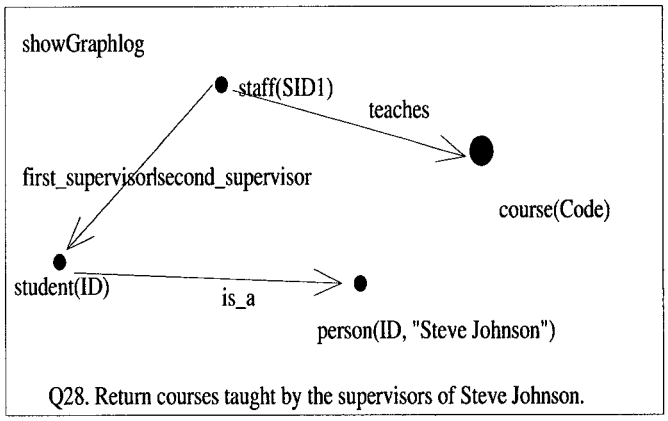


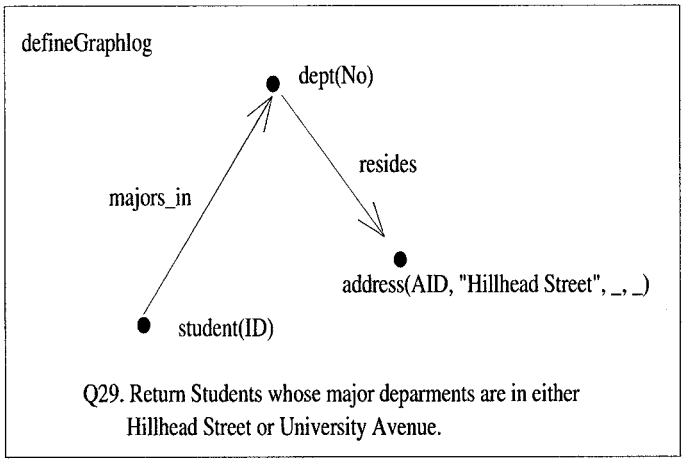
Figure 63: Diagrams of Example Query Composition(23 – 26)



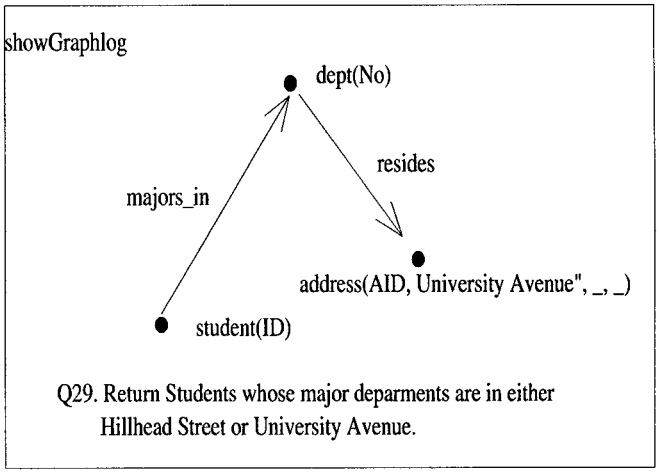
Q27.Return students supervised by Steve Johnson.



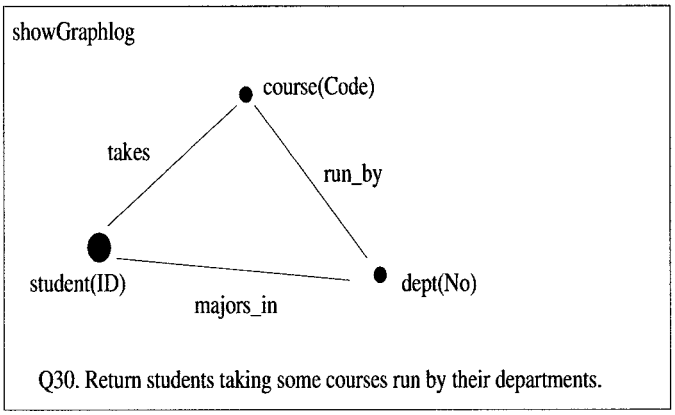
Q28. Return courses taught by the supervisors of Steve Johnson.



Q29. Return Students whose major deparments are in either Hillhead Street or University Avenue.



Q29. Return Students whose major deparments are in either Hillhead Street or University Avenue.



Q30. Return students taking some courses run by their departments.

Figure 64: Diagrams of Example Query Composition(27 – 30)