bcnQL: A query language for biochemical networks

H. Yang and R. Sunderraman*

Department of Computer Science,
Georgia State University,
Atlanta, GA 30303, USA
E-mail: hyang9@student.gsu.edu
E-mail: raj@gsu.edu
*Corresponding author

H. Tian

Chronic Viral Diseases Branch Centers
for Disease Control and Prevention,
1600 Clifton Road,
Atlanta, GA 30333, USA
E-mail: htian@cdc.gov

Abstract: This paper proposes a graph data model that can represent information present in Biochemical Networks. The study presented in this paper also proposes a query language, called bcnQL, which empowers users to query entities, interactions, processes and pathways with arbitrary conditions. We then discuss the query-processing techniques, more specifically, the translation of bcnQL queries into G-algebra and a set of algebraic operators on graph objects. Some query examples are presented to demonstrate the applicability of the language for this specific domain. Finally, we provide details of a prototype implementation for the query language.

Keywords: graph data model; query language; G-algebra; biochemical networks.


Biographical notes: Hong Yang received her PhD in Computer Science from Georgia State University in 2009. Her research interests include database applications in bioinformatics and data mining.

Rajshekhar Sunderraman is Professor of Computer Science at Georgia State University in Atlanta, Georgia. His research interests are in deductive databases, logic programming, incomplete and inconsistent information, semi-structured data and data modelling of biological data. He has published over 100 research papers in leading computer science journals and conferences and is the author of a leading textbook “Oracle 10g Programming: A Primer” published by Addison Wesley. He received the BE (Honours) in Electronics Engineering from BITS Pilani in 1980, the MTech in Computer Engineering from IIT Delhi in 1982, and the PhD in Computer Science from Iowa State University in 1988.
1 Introduction

The conventional perspective for investigating genomic information often relies on the study of DNA sequence (Venter et al., 2001). However, DNA sequence is not sufficient to reflect the actions of genes. The protein products of these genes assemble at appropriate cellular locations to coordinate their biological functions (Krishnamurthy et al., 2003). Thus, to better study the information of such genes, biochemical network is considered as an alternative to DNA sequence.

Biochemical networks are generally subdivided into three types: metabolic, regulatory and signal transduction networks. Metabolic networks describe proteins, genes, reactions, etc., and focus on the way matter flows in cells. In regulatory networks, the focus is on the way different controls affect each other, such as catalysis of a reaction, regulation of the expression of a gene and inhibition of a catalyst. Signal transduction networks focus on the transfer of information from the extracellular medium to the cell nucleus.

Nowadays, the study on biochemical networks attracts many researchers attention and as a result a large number of databases have emerged to store the biochemical networks data. ENZYME, a satellite database of SWISS-PROT, contains information about Enzyme Classification (EC), cofactors and catalytic activity for each type of characterised enzyme (Bairoch, 2000). KEGG is a genome database that contains information on proteins and enzymes whose gene sequences have been determined (Kanehisa and Goto, 2000). The Bind database focuses on protein interactions (Bader et al., 2001). CSNdb focuses on signalling pathways (Takai-Igarashi et al., 1998). The previously introduced databases are dedicated to one particular network type. However, EcoCyc (Karp, 2000) and aMAZE (van Helden et al., 2000) use data models to integrate a variety of processes, including metabolic pathways, regulation of gene expression and enzyme function, and signal transduction and transport. The biochemical networks in these integrated models can be viewed as complex and typed graphs.

To analyse such network databases, there is an urgent need for a system that supports ad hoc querying. Some typical queries against a pathway database are listed here:

- find all reactions involving a certain substance
- find all processes that lead from node A to node B in less than X steps and more than Y steps
- find the shortest path between two substances that includes a third substance
- find all processes within a given number of steps from a process in a pathway, often referred to as the neighbourhood problem.
Some projects provide a set of predefined queries as those listed earlier. PathCase (Krishnamurthy et al., 2003) provides predefined queries executed through a dialogue box for a particular pathway, process, or a molecule entity. aMAZE (van Helden et al., 2000) contains a metabolic-path-finding tool that is able to find the shortest path between two nodes. KEGG provides extensive visualisations of the pathways with EC numbers, however, it does not support the dynamic querying. BioCyc encompasses a query page that allows users to query pathway databases by specifying conditions on the entity’s name. In summary, the tools available for querying the information stored in these pathway databases are restricted to the full-text search for node names and the search for the paths between two given nodes. They do not support conditions on paths or graphs that are often required in practice. In addition, the programming effort to process queries for one pathway database system cannot be shared with another. We believe that a declarative language with the following features is strongly needed for biochemical network databases.

- The data model should reflect the features of biochemical pathways and should support metabolic pathways, regulatory pathways and signal transduction pathways.
- The data model should be general enough to be used by different biochemical network databases.
- The query language should be capable of querying biological entities, interactions, processes and pathways and should support query conditions. Therefore, the query language should be able to describe and represent various types of processes and pathways that take place in a cell.
- In addition, the bcnQL should be capable of constructing new graphs, which may not be presented in the database with defined graph operations.

In this paper, we propose bcnQL, a graph query language for biochemical networks to meet the above-mentioned criteria. The proposed graph data model is an object-oriented data model, which is extended from the data model designed in van Helden et al. (2000). This data model encompasses Node class to represent the biochemical entities and interactions, Edge class to represent the relationship between nodes, Path class to represent various types of processes and Hyper-Path class to represent the pathways. Another major feature of the language is that it can also be used to compose interesting sub-graphs with hyper-paths. The graph class is used to represent/store such sub-graphs. Queries in the proposed query language are constructed using three clauses: output, define and where clauses. Each clause provides a necessary functionality that the query language needs to fulfil.

The rest of the paper is organised as follows. In Section 2, some related works are reviewed and their disadvantages are discussed. In Section 3, the biochemical networks data model is introduced. In Section 4, the syntax and semantics of the proposed query language are discussed. Section 5 discusses the query-processing techniques for bcnQL. In Section 6, several query examples for existing biochemical network database systems are represented in bcnQL to evaluate its capabilities for biochemical networks queries. The prototype implementation for the query language is discussed in Section 7. Section 8 concludes the paper.
2 Related work and comparison

Graph databases have been an area of research in the database community for some years and attracted many researchers’ interests. There are some existing query languages for databases applications in which a graph structure can be used to naturally represent the database.

Gram (Amann and Scholl, 1993) is a graph query language system in which data is organised as graphs in which each node represents a value specified in the corresponding domain and each edge represents a relation between the nodes. A main feature of the language is the use of the regular expression for explicit definition of paths called walks; a set of walks can be combined and is referred to as hyper-walks. In addition, algebraic operations on hyper-walks including selection, projection, renaming, join, concatenation and various set operations are defined. However, one major problem of this language is that conditions cannot be specified on the properties of walks or hyper-walks thereby limiting the expressive power of the language.

GraphDB (Guting, 1994) consists of a data model and a query language that is able to model and query graph data for spatial application. The data model is an extension of an object-oriented data model with simple class, link class and path class representing nodes, edges and paths of the database graph, respectively. To perform a query, the “on … where … derive” statement is available to construct a set of simple, link/path objects. A ‘rewrite’ operator is used to retrieve objects of interest. However, this language does not provide enough capability to query some sub-graphs of interest. In GraphDB, the entities (node, edge and path) that compose the queried sub-graph or the entities that are not included in the queried sub-graph need to be identified to be able to query the sub-graph. As a result, GraphDB explicitly lists all the entities that are or are not elements of the sub-graph. However, the relationships among the entities are not given, which prevents the GraphDB from being a flexible language in querying sub-graphs.

GraphQL (He and Singh, 2007) is a query language supporting graph queries with the use of a graph pattern, which is the basic operational unit of the query. A graph pattern consists of graph structure and a predicate that allows constraints to be specified on nodes, edges, or graphs. The paper also defines a graph algebra including selection, composition, projection and join operators that are closed on graphs. In addition, access methods for the selection operator are addressed in this language. However, queries involving relative ordering of sequence elements cannot be conducted.

GOQL (Sheng et al., 1999) consists of an object-oriented graph data model and is essentially a graph query language (GOQL) extended from OQL to be able to query multimedia graphs. GOQL provides operators such as next, until and connected for querying paths and sequences. However, this language cannot adequately meet the requirements for sub-graph querying.

PQL (Leser, 2005) is a query language proposed for biological networks. PQL is extended from a simple graph data model to reflect properties of biological objects. Graph match queries are conducted using the information associated with the node properties and paths between the nodes. However, the properties of paths and graphs cannot be specified by the language, but they are often necessary to be represented to effectively perform graph query.
As discussed in the introduction section, the query tools in existing Biochemical Systems (Bairoch, 2000; Bader et al., 2001; Karp, 2000; Kanehisa and Goto, 2000; Krishnamurthy et al., 2003; Takai-Igarashi et al., 1998; van Helden et al., 2000) are restricted to the full-text search for node names and the search for the paths between two given nodes. For these simple queries, they do not allow conditions on paths or graphs that are often required in practice. In addition, the programming effort to process queries for one pathway database system cannot be shared for another.

3 Graph data model

A general data model that is used to represent Biochemical Networks, which integrates metabolic, regulatory and signal transduction information, is described in aMAZE (van Helden et al., 2000). The data model in aMAZE contains two main object types: BiochemicalEntities and BiochemicalInteractions. The BiochemicalEntities objects are molecules of a biochemical network that could be proteins, compounds, genes, mRNA, enzymes, etc., and the attributes describing their physical characteristics are associated with those entities. The BiochemicalInteractions objects represent relationships between nodes and the relationships could be transformations in which a set of molecules are transformed into another set of molecules such as metabolic process/pathway or controls in which there is a set of molecules as inputs and other interactions as outputs such as regulatory pathway.

Biochemical Networks can be represented by a graph in which the node can be either an entity or an interaction and the arrows represent the input/output relationships between nodes (Figure 1). We extend the data model described in aMAZE to a graph model with Edge, Path, Hyper-Path and Graph classes. Thus, the data model has features of the object-oriented model to represent integrated networks. The classes and their properties are listed in Figure 2.

Figure 1  A small biochemical network

In the Node class, each Node object has a unique name, which is the denotation on the node in the graph. The nodes in the graph may be different data types, and each data type could be represented by an extended subclass of the Node class. The BioEntity_Gene, BioEntity_RNA, BioEntity_Protein, BioEntity_Compound and BioInteraction are subclasses inherited from the Node class.
In the Hyper-path class, property ‘paths’ refers to a set of Path objects that constitute the Hyper-Path object. There are different types of processes or pathways existing in the Biochemical Networks. Path objects can be used to represent these processes. A collection of processes constitute a pathway, therefore, Hyper-Path objects can represent the pathways.

4 Biochemical network query language

In this section, the proposed Biochemical Network query language is presented, and this language will be referred to as bcnQL hereafter.

A simple bcnQL query consists of five clauses: Output clause, Define clause, Where clause, Group by clause and Order by clause. The simple bcnQL can be combined through the set operators like UNION, INTERSECT and MINUS. The form of a simple bcnQL is illustrated here:

```
Output  <Output-list>
Define   <Node-formulas>| <Edge-formulas>| <Path-formulas>| <Hyper-path-formulas>| <Graph-formulas>
[Where   <selection-predicate>]  
[Group By <grouping-objects-attributes>] 
[Order By <ordering-attributes>]
```

The ‘Define’ clause allows users to define one or more variables using node, edge, path, hyper-path or graph formulas, and most of the ‘expressibility’ of the language comes
from the Define clause, the ‘Output’ clause may output objects, their properties (dot-path expressions), or aggregates, the ‘Where’ clause is used to express selection conditions, the ‘Group by’ clause allows the objects to be grouped based on certain attributes, and the ‘Order by’ clause allows the objects to be ordered by certain attributes. The bracketed expressions are optional.

The informal semantics of a simple bcnQL query is described as follows:

2. The tuples of objects that satisfy the selection-predicate are retained.
3. If the ‘Group by’ clause is present, the results are split into partitions based on partition attribute(s).
4. If the ‘Order by’ clause is present, the results are sorted by the attribute defined in the ‘Order by’.
5. Output clause expressions are evaluated as query results.

In the following subsections, each formula present in the ‘Define clause’ will be described. In Section 4.1, we will present how to select the nodes and edges satisfying certain conditions specified in the formulas. In Section 4.2, we will discuss how to use the path formula to produce a collection of path objects. An explanation of generating a collection of hyper-path objects utilising the hyper-path formula will be given in Section 4.3, followed by the discussion on how the graph objects are composed by the hyper-path objects in Section 4.4.

4.1 Node/edge formulas

The form of the node and edge formula is given by:

\[
\text{[ (NodeTypeClass) <predicate> ]}
\]

and

\[
\text{((EdgeTypeClass) <predicate> )}
\]

respectively, where

- **NodeTypeClass** includes Node class and all the extended Node classes, which provide the ability to work with specific applications
- **EdgeTypeClass** includes Edge class and all the extended Edge classes, which provide the ability to work with specific applications
- **<predicate>** is a selection condition on the attributes of the Node/Edge objects.

If (subclass) is specified, its objects satisfying the predicate are returned; otherwise, Node or Edge objects satisfying the predicate are returned.
4.2 Path formulas

Here, syntax for simple path formulas, simple path formulas with regular expression and general path formulas are given.

4.2.1 Simple path formulas

Paths

[(NodeTypeClass)<predicate>]  
((EdgeTypeClass)<predicate>)  
[(NodeTypeClass) <predicate>] as p1

This formula defines a variable p1 that denotes all paths in the graph database, which originate at a node satisfying the first node predicate and is connected by an edge (satisfying the edge predicate) to another node that satisfies the second node predicate.

4.2.2 Simple path formulas with regular expression

Paths

[(NodeTypeClass) <predicate>]  
_*  
[(NodeTypeClass) <predicate>] as p1

The * in this formula may represent an edge or a path. This formula defines a variable p1 that denotes all paths in the graph database, which originate at a node satisfying the first node predicate and terminate in a node satisfying the second node predicate. If a sub-class name is specified in the first node formula, all in-between nodes are assumed to have the same object type as the first node.

In the previous formula, * can be replaced by either an integer N or character ‘S’ to denote a path of fixed length N or shortest path, respectively, as indicated here:

Paths

[(NodeTypeClass) <predicate>]  
_*N  
[(NodeTypeClass) <predicate>] as p1

Paths

[(NodeTypeClass) <predicate>]  
_*S  
[(NodeTypeClass) <predicate>] as p1
4.2.3 General path formulas

The general path formula is formed by concatenating one or more simple path-formulas by introducing edges between them as indicated here:

\[ p_1 \ ((\text{EdgeTypeClass} \ <\text{predicate}>)) \]
\[ p_2 \ ((\text{EdgeTypeClass} \ <\text{predicate}>)) \]
\[ \ldots \]
\[ \ldots \]
\[ p_{n-1} \ ((\text{EdgeTypeClass} \ <\text{predicate}>)) \]
\[ p_n \]

4.3 Hyper-path formulas

Hyper-path formulas have the following form:

\[ \text{Hyper-path} \]
\[ (\text{<path-formula> p}_1 + \ldots + \text{<path-formula> p}_n) \]

where \[\text{graph-predicate}\]

In this expression, a graph object is constructed by a collection of paths based on the \[\text{graph-predicate}\] that specifies the connecting points in the paths. The path from each \[p_i\] is chosen and connected using the \[\text{graph-predicate}\].

4.4 Graph formulas

Graph formulas are used to define graph objects, which are generated by graph functions. The graph functions include union and intersect.

4.4.1 Union function

\[ \text{Union} (\text{<hyper-path-formula> AS h}_1, \text{<hyper-path-formula> AS h}_2) \]

The union function encompasses two hyper-path arguments as shown in the above-mentioned formula. The result of the union function is a graph that combines the two hyper-path arguments, where all of vertices and edges are unique.

4.4.2 Intersect function

\[ \text{Intersect} (\text{<hyper-path-formula> AS h}_1, \text{<hyper-path-formula> AS h}_2) \]

The intersect function returns the graph that is present in both hyper-path arguments.
5 bcnQL query processing

bcnQL query expressions are translated into G-algebra, a set of operators that manipulate graph objects, for query processing. These algebra operators can be divided into two groups: core algebra operators and paths/hyper-paths algebra operators. Here, we will give semantics for each operator.

5.1 Core algebra operators

Core algebra operators are similar to relational algebra, including Select ($\sigma$), Project ($\Pi$), Cartesian product ($\times$) and Join ($\bowtie$).

5.1.1 Select

The select operator is defined as follows:

$$\sigma_{\rho}(C) = \{c | c \in C \text{ and } \rho \text{ is true}\}.$$  

Here, $C$ is one of the class types defined in graph data model, i.e., $C$ could be node type, edge type, path type, hyper-path type, graph type or other extended types. $\rho$ denotes the set of Boolean combinations of predicates. The inputs of the select operator are $C$ and $\rho$, and the output is a collection of objects $c$ of class $C$, which makes the value of $\rho$ to be true.

5.1.2 Cartesian product, join

A Cartesian product operator ($\times$) takes two collections of path objects and produces one collection of hyper-path objects as output.

The join operator ($\bowtie$) can be defined by a Cartesian product followed by a selection:

$$A(\bowtie)B = \sigma_{\rho}(A \times B)$$

where $\rho$ denotes the set of Boolean combinations of predicates.

5.1.3 Project

The project operator ($\Pi$) selects certain attributes from the object and discards other attributes. If we are interested in only certain attributes of an object, we use the Project operation to project the object over these attributes. Let $C$ be a collection of objects, to list the $A_1, A_2, \ldots, A_n$ attributes of $C$, the Project operation can be expressed as:

$$\Pi < A_1, A_2, \ldots, A_n > (A).$$

5.1.4 Set operators

The union operator unions the elements of two sets and the same elements in both sets are only shown once in the new set. Similarly, the intersect operator returns the common elements in the two sets.
5.2 Path and hyper-path algebra operators

These operators are used to implement the define clause of the bcnQL query for producing collection of objects. $\mathcal{O}_{\text{until}}$, $\mathcal{O}_{\text{limitedLength}}$, $\mathcal{O}_{\text{shortest}}$, $\mathcal{O}_{\text{next}}$ and $\mathcal{O}_{\text{connect}}$ are used to denote the path and hyper-path algebra operators. The semantics of these operators are described as follows:

5.2.1 Until operator

Until operator takes two collections of node-type objects from class $X_1$ and $X_2$, respectively, and produces a collection of Path objects as output.

$$\mathcal{O}_{\text{until}}(X_1, X_2, G) = \{ p | p \text{.start} = x_1 \text{ and } x_1 \in \sigma_{\text{pl}}(X_2) \text{ and } p \text{.end} = x_2 \text{ and } x_2 \in \sigma_{\text{pl}}(X_2) \}.$$ 

That is, for each pair of nodes $x_1$, where $x_1 \in \sigma_{\text{pl}}(X_1)$ and $x_2 \in \sigma_{\text{pl}}(X_2)$, if there exists a path starts with $x_1$ and ends with $x_2$ in graph $G$, then a path object is generated.

5.2.2 Limited length operator

Limited length operator is similar to Until operator except that Limited length operator outputs a collection of Path with a specified length.

$$\mathcal{O}_{\text{limitedLength}}(X_1, X_2, G) = \{ p | p \text{.start} = x_1 \text{ and } x_1 \in \sigma_{\text{pl}}(X_1) \text{ and } p \text{.end} = x_2 \text{ and } x_2 \in \sigma_{\text{pl}}(X_2) \text{ and } p \text{.length} = \text{LimitedLength} \}.$$ 

5.2.3 Shortest operator

Shortest operator takes two collections of node-type objects from class $X_1$ and $X_2$, respectively, and produces a collection of the shortest Path between $x_1$ and $x_2$ as output.

$$\mathcal{O}_{\text{shortest}}(X_1, X_2, G) = \{ p | p \text{.start} = x_1 \text{ and } x_1 \in \sigma_{\text{pl}}(X_1) \text{ and } p \text{.end} = x_2 \text{ and } x_2 \in \sigma_{\text{pl}}(X_2) \text{ and } p \text{.length} = \text{shortest} \}.$$ 

5.2.4 Next operator

Next operator takes two collections of node-type objects from class $X_1$, $X_2$ and one collection of edge-type objects from class $Y$ as input, and produces a collection of path objects as output.

$$\mathcal{O}_{\text{next}}(X_1, X_2, Y) = \{ p | p \text{.start} = y \text{.from} = x_1 \text{ and } p \text{.end} = y \text{.to} = x_2 \text{ and } x_1 \in \sigma_{\text{pl}}(X_1) \text{ and } x_2 \in \sigma_{\text{pl}}(X_2) \text{ and } y \in \sigma_{\text{pl}}(Y) \}.$$ 

That is, for each pair of nodes $x_1$ and $x_2$, where $x_1 \in \sigma_{\text{pl}}(X_1)$ and $x_2 \in \sigma_{\text{pl}}(X_2)$, if there exists an edge $y$ where $y \in \sigma_{\text{pl}}(Y)$ connects nodes $x_1$ and $x_2$, then a path object starting with $x_1$ and ending with $x_2$ is generated.
5.2.5 Connect operator

Connect operator takes two collections of path objects from $P_1$ and $P_2$ as input, and produces a collection of hyper-path objects as output. Connect operator must work together with predicate $\rho$, which gives the connection condition.

$$O^\text{connect}_\rho(P_1, P_2) = \{ h \mid h \in P_1 \text{ join } P_2 \text{ and } \left( \begin{array}{l} h.\text{nodes} = p_1.\text{nodes} U p_2.\text{nodes} \text{ and} \\ h.\text{edges} = p_1.\text{edges} U p_2.\text{edges} \end{array} \right) \text{ and } p_1 \in P_1 \text{ and } p_2 \in P_2 \}.$$  

6 Biochemical network query examples

In this section, some biochemical network query examples collected from existing biochemical network systems aMAZE (van Helden et al., 2000), Pathcase (Krishnamurthy et al., 2003) will be discussed to demonstrate the sufficiency of bcnQL for biochemical network application. In addition, one of them is used to show how the bcnQL is translated into G-algebra defined in the previous section.

**Example 1:** Find connectivity of the reactions in the network. This query can be expressed in bcnQL as follows:

Output `count(p1.start), count(p2.end)`

Define `[(Node)]_1[(BiochemicalInteraction)]_1[(Node)]` as `p1`

Group by `p1.nodes.elementAt[2]`

One area that biochemists are interested in is the global structural properties of the network, such as average connectivity of the reactions in the network. First, we calculate the connectivity of each reaction in the network, and then the average connectivity of the reactions is calculated by averaging the number of inputs and number of outputs.

**Example 2:** Find all processes that lead from compound A to compound B in less than 5 steps, and more than three steps in the global network. Figure 3 gives a graphical view of this query.

**Figure 3** Query example 2
The above-mentioned query can be expressed in bcnQL as follows:

Output p1
Define path [(Node) name='A']_*[(Node) name='B'] as p1
Where p1.length>=3 and p1.length <=5

Two end path finding can be used to discover alternative pathways to the classical pathway. Finding all possible paths between two compounds requires a lot of calculation time and also it may not always produce a practical solution because the long path may lose their biochemical meaning. Therefore, the path length constraints are added to produce meaningful results. The bcnQL is translated into G-algebra by the following steps:

- Translate node expression [(Node) name= 'A'] into select operator: σname='A' (Node)
- Translate node expression [(Node) name= 'B'] into select operator: σname='B' (Node)
- Translate the wildcard path search into until operator: σρ1 =(name='A') and ρ2=(name= 'B') (Node, Node, BiochemicalNetwork)
- Bind the result to an iterator variable p1: σρ1 =(name='A') and ρ2=(name= 'B') (Node, Node, BiochemicalNetwork) = {p1|p1.startσname='A' (Node) and p1.endσname='B' (Node)}
- Finally output path objects through select operator: σp1.length>=3 and p1.length <=5 (path)

Example 3: User specifies a set of nodes in the network – the ‘seed’ nodes – and prompts the system to extract the portions of the network or sub-graphs that interconnect each pair of seed nodes via the smallest number of individual links (Note: here we use three nodes (A, B, C) as the ‘seed’ nodes)). The above-mentioned query request can be expressed in bcnQL as follows:

Output h1
Define hyper-path
{(Node) name='A' _S [(Node) name='B'] as p1 +
(Node) name='A' _S [(Node) name='C'] as p2 +
(Node) name='B' _S [(Node) name='C'] as p3)}
where [p1.start =p2.start and p1.end = p3.start and
p2.end = p3.end]
as h1

Besides path finding between two given nodes to find the alternative pathways, biochemists can use pathway reconstitution from clusters of reactions as the starting point for building alternative pathways. To build the pathways is starting from using some reactions as ‘seeds’, and the procedure is to interconnect the seeds through the shortest path. The returned reconstituted pathway is a sub-graph containing all the seed nodes, as well as the arrows and intermediate nodes traversed from one seed to the other seed.
7 Prototype implementation

In this section, the bcnQL query system is presented as a proof of implementation. bcnQL is implemented on the top of an object database, db4o (http://www.db4o.org). Db4o is an object database that is specifically designed for providing persistence for object-oriented programmes and stores objects.

The evaluation of bcnQL queries consists of four steps. First, the classes defined in graph data model are transformed into db4o classes through the use of Class Creator. Second, the graph objects are stored in db4o using Object Storer. Then, the bcnQL query expressions are sent to the bcnQL parser (implemented using JFlex/JCUP) in which the syntax and semantics of the query are verified. In the last step, upon passing the syntax and semantic checks, the query is translated into G-algebra for query processing. Figure 4 illustrates the architecture of the query system.

Figure 4  Architecture of the query system

7.1 Mapping graph data model to Db4o classes

With the use of object database db4o, the database schema matches the graph data model very closely. The five basic classes – Node, Edge, Path, Hyper-path and Graph, can be simply mapped into five db4o classes. These are predefined in bcnQL query system. Besides the five basic classes, the application-specific extended classes defined in the graph data model can also be mapped into db4o classes by using Class Creator.
7.2 bcnQL Parser

To validate the bcnQL query expressions, the syntax and semantics of the language need to be verified. Syntax refers to the way by which symbols are combined to create well-formed expressions of the language. Syntax defines the formal relations between the components of a language, thereby providing a structural description of various expressions that make up legal strings in the language. Semantic refers to the meaning of the symbols arranged with that structure. bcnQL parser is designed to validate the bcnQL query expression. It includes the following operations: lexical analysis, syntactic analysis and semantic analysis. Figure 5 demonstrates how to validate the bcnQL query expressions using the bcnQL parser.

The first stage of the process is lexical analysis, or scanning, by which the input character stream is split into meaningful tokens. For example, a node formula \([(a)b > 0]\) can be split into the tokens \[, (, a, ), b, >, 0 and \]. The next stage is syntactic analysis, which is checking that the tokens form an allowable expression and building a parse tree. The final phase is semantic analysis, which adds semantic information to the parse tree to replace the linear sequence of tokens with a tree structure built based on the rules of a formal grammar that define the language’s syntax. This phase performs semantic checks such as type checking (check for type errors), operands and operator match checking, class or attribute name existing checking and issuing warnings/errors.

Figure 5 bcnQL Parser process
JFlex (http://www.jflex.de) and Java™ Based Constructor of Useful Parsers (CUP for short) (http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html) are used to implement the bcnQL parser. JFlex is a lexical analyser generator (also known as a scanner generator) written in Java. JCUP is a system for generating look-ahead LR parser, which checks for correct syntax and builds parse tree.

7.3 bcnQL evaluator

After the bcnQL query expression passes the syntax and semantic checks, the query expressions need to be evaluated and results have to be printed. bcnQL evaluator translates bcnQL queries into G-algebra consisting of two groups of operators introduced in Section 3. db4o supports three types of query mechanisms: Query-By-Example (QBE), Native Queries and Simple Object Data Access (SODA). QBE is the basic query mechanism offered by db4o. It works well when retrieving objects with one or more specific attributes, but it becomes inept when more complicated queries need to be performed on the database. Both Native Queries and SODA can be used for complex queries, whereas SODA is db4o’s low-level querying API used for dynamic generation of queries. In the implementation, SODA API is chosen to implement the core algebra operators discussed earlier. The second group includes algebra operators for querying paths and hyper-paths. The algorithms for implementing these operators can be found in Yang and Sunderraman (2008).

8 Conclusions

This paper presents an object-oriented graph data model, which can be used to represent Biochemical Networks including metabolic, regulatory and signal transduction networks. This data model includes Node class and extended Node subclasses to represent Biochemical Entities and Interactions, Edge class to represent relationship between nodes. Path and Hyper-Path classes are proposed to represent various processes and pathways and Graph class is used to represent graph objects, which may be present in the database or new graphs objects that are composed with hyper-paths.

The paper also proposes a query language, bcnQL, which empowers users to query entities, interactions, processes, pathways with conditions specified on them. In addition, bcnQL could be used for composing interesting graphs with defined graph operations. The query system prototype is implemented, and the employed algorithms show that bcnQL is practical and efficient.

References


Yang, H. and Sunderraman, R. (2008) bcnQL: A Query Language for Querying Graph Data, Manuscript, Department of Computer Science, Georgia State University.

Websites


JFlex – The Fast Scanner Generator for Java: http://www.jflex.de/