Query Language for Prometheus

Cédric Raguenaud, Jessie Kennedy, Peter J. Barclay

Database and Object Systems Group
School of Computing
Napier University
219 Colinton Road
Edinburgh EH14 1DJ
United Kingdom
{cedric, jessie, pete}@dcs.napier.ac.uk

Abstract:
This document presents the query language developed for the Prometheus database [RKB99]. This language is based on graphs and provides the user with a means to define the data to be extracted from the database and its manipulation. The language requires the definition of patterns that can be matched against the set of data available in the database and return instances.

1. Introduction

We have defined in [RKB99] a database model based on graphs and that puts the emphasis on relationships. This model was designed to support the Prometheus taxonomic model [PK99] and provide taxonomists with means of manipulating taxonomic data naturally and effectively. Since taxonomists are not computing specialists, the choice of graphs also has the advantage of being naturally represented graphically and easily coupled with a graphical programming language (e.g. [PL94], [RK97]).

First we define briefly the Prometheus model in section 2, then we explain the operations necessary on graphs in section 3. In section 4 we describe the structure and the properties of our patterns and in section 5 we build a database example and show how it works. Finally we conclude in section 6.

2. The Prometheus model

The Prometheus model is based on directed graphs. First we define a generic graph and then we explain how it is derived in order to create our model.

2.1. Generic graph

A graph is constituted of nodes and arcs. Nodes represent basic concepts such as numbers or the core of a complex object such as an NT (as shown in figure 1 in a simplified form), and arcs are directed edges that bind two nodes by a named relationship (the label of the arc carries the name of the relationship). Both nodes and arcs are labelled and their labels are unique. This label serves as unique identifier. The node where an arc starts is called the source of the arc, and the node where the arc ends is called the destination of the arc.

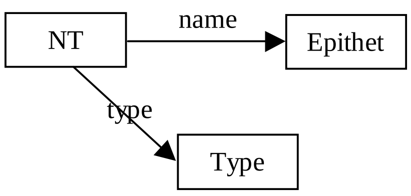


Figure 1: Example of graph

In this example, an NT node is involved in two relationships (represented by arcs), one with an Epithet node, and one with a Type node.

More formally:
A directed graph G is G = (N, A, label, source, dest) where N is a set of nodes, and A is a set of arcs. A node $n \in N$ is identified by its label $nl \in (NL \subseteq L)$ which is unique in all elements of the graph. A directed edge or arc $a \in A$ is defined by (sl, el, dl), where $sl, dl \in NL$, $al \in (AL \subseteq L)$ is a unique label in all elements of the graph. sl is called the source of the arc, and dl is called the destination. We also have $NL \cap AL = \emptyset$ and $N \cap A = \emptyset$. We define three operations: label(o): $o \in (N \cup A) \rightarrow L$, which returns the label of an object $o \in (N \cup A)$; source(a): $a \in A \rightarrow nl \in NL$ which returns the label source for the arc $a \in A$; dest(a): $a \in A \rightarrow nl \in NL$ which returns the label destination for the arc $a \in A$.

Like in classical graph theory, we say that two nodes n_1, n_2 related by an arc $a_1(n_1, n_2)$ are adjacent, and that the arc a_1 is incident from n_1 and incident to n_2. The sum of the number of arcs incident from (positive degree, noted pos $d(n_1)$) and to (negative degree, noted neg $d(n_1)$) a given node is called the degree of that node.

These are the basic structures understood by the database, and all other concepts are derived from them.

Nodes are of two kinds: atomic nodes and complex nodes. Nodes that have a purely negative degree are called atomic nodes. They are complex nodes otherwise. Atomic nodes are used to represent basic concepts such as numbers or character strings. In that case, their label is called the value of the node. Hence, the positive degree of an atomic node is always zero, i.e. if n_1 is an atomic node, pos $d(n_1) = 0$.

We used indirect references to nodes using labels because these labels, in addition to being the values of atomic nodes, fulfil also the function of unique identifiers. Our system does not support object identity in the sense of object-oriented databases.

2.2. Model

Our graphs must first be extended to represent some more complex data, such as complex objects. Thus we first divide both nodes and arcs into different categories: we define atomic nodes (which have no outgoing arcs) and composite nodes (that might have outgoing arcs), simple arcs and multiple arcs, associations and aggregations (which are subject to interpretation by the database engine. In fact, we define many sets of entities.

This definition of graphs is extended in order to capture types and instances. Types are necessary for guiding the user in its interaction with the database. Although the database is designed for supporting taxonomic work and this work is largely free of constraints, specific areas of taxonomic work require the application of the nomenclatural code. Therefore, our data must follow a schema and implement integrity check. Our graphs are therefore directed and typed. A type for a node means that all instances of the type must be involved at most in the same relationships as their type. For an arc, it means that it originates and ends in node instances of the type that its type groups together. A type restricts the data that can be input in the database and is the first step towards integrity checking. In addition, we introduce the concept of synonymy at type and instance level. At type level, synonymy allows a grouping of types under a common meaning, but unlike in an object-oriented environment, it does not imply the sharing of structure or behaviour. A synonym of types, also called union type, only means that all types grouped under it can be involved indifferently in some relationships. At instance level, a synonym means that the instances grouped under it represent the same real world object, although they may contain different values. It is extremely useful to represent databases of persons that might have their name spelled differently, but are still the same real world person.

A formal definition can be found in [RKB99].

3. Graph modification.

The operators we define here only affect the lowest part of the database. They are directly applied on the graphs defined by the user. At this stage, we only need to define the basic operations required by a graph structure: adding and deleting nodes and arcs.

4 operations are defined on the graph database: add node, add arc, delete node, and delete arc. Each of these operations is specialised in type and instance operations:

- Add type node $tn_1 \in TN$ with label $l_1 \in TNL$: creates a type node tn_1 and assigns the label l_1 to it if it does not already exist.
- Add union type node $tun_1 \in TUN$ with label $l_1 \in TUNL$ and types $\{tn_i\} \in TCN \cup TUN$: creates a union type node tun_1, assigns the label l_1 to it if it does not already exist, and sets the types unioned in this union type by creating necessary type arcs (with a unique computer generated name).
- Add instance node $in_1 \in IN$ from type node $tn_1 \in TN\text{-}TUN$: creates a new instance node conforming to the type node tn_1 and assigns a unique computer generated label to it.
- Add type arc $ta_1 \in TA$ with label $l_1 \in TAL$, source $tn_1 \in TN\text{-}TAN$ and destination $tn_2 \in TN\text{-}TSN$: create a type arc ta_1, assigns the label l_1 to it if it does not already exist, and sets the source and destination of the arc.
- Add instance arc $ia_1 \in IA$ from type arc $ta_1 \in TA$ with $source(ia_1) = label(in_1) \wedge source(ia_1) = label(in_2)$ where in_1, $in_2 \in IN$: creates a new instance arc ia_1, assigns a unique computer generated label to it, and sets its source and destination. ia_1 must conform to the type arc ta_1. If in_1 is involved in an instance synonymy relationship, all the instances involved in this relationship are subjected to the same arc addition if they do not already exist, otherwise it is rejected as incompatible.
- Delete type node $tn_1 \in TN$: deletes the type node tn_1 and all type arcs from which it is the source or the destination, i.e. $\forall\ ta \in TA$, $((source(ta) = label(tn_1)) \vee (dest(ta) = label(tn_1))) \Rightarrow delete(ta)$. All instances of the type are also deleted, i.e. $\forall\ in_1 \in IN$, $type(in_1) = label(tn_1) \Rightarrow delete(in_1)$. The type is actually deleted only if no other object in the system references it.

- Delete instance node $in_1 \in IN$: deletes the instance node in_1 and all instance arcs from which it is the source or the destination, i.e. $\forall$ ia $\in$ IA, ((source(ia) = label(in_1)) $\vee$ (dest(ia) = label(in_1))) => delete(ia). Like for the arc addition, if in_1 is involved in an instance synonymy relationship, all the instances involved in this relationship are subjected to the arc deletion. The instance is actually deleted only if no other object in the system references it.
- Delete type arc $ta_1 \in TA$: deletes the type arc ta_1. All arc instances of this type are deleted, i.e. $\forall$ ia $\in$ IA, type(ia) = label(ta_1) => delete(ia).
- Delete instance arc $ia_1 \in IA$: deletes the instance arc ia_1. If ia_1 represents a concept of aggregation (i.e. $ia_1 \in IAA$), dest(ia_1) must be deleted, i.e. $\forall$ in $\in$ IN, dest(ia_1) = label(in) => delete(in).

Each time an instance is created (be it an arc or a node), the type conformation algorithm is applied in order to make sure that the structure created is compatible with the schema from which it is instantiated. If it is not, it is rejected and an error is signalled to the user.

4. Queries.

In addition to these basic operations, we need a mechanism that allows the user the retrieval of data from the database and the specification of the emplacement of modifications. For the purpose of taxonomic work, the retrieval facility of our language must be able to extract the extent of a type (i.e. all instances of this type), combinations of instances obeying a template (e.g. a NT and its Author), and recursive queries (e.g. extracting a classification). Our aim is not to create a computationally complete language for the sake of it.

The simplest way from a user point of view is the definition of a pattern that will be matched against the database and return a subgraph of it. A pattern, or template, is the representation of the structure searched in the database.

4.1. Pattern

A pattern is a graph defined as previous graphs, but it is constituted of representative objects. These objects represent nodes that can actually be found in the database (i.e. types and instances). Unlike type and instance graphs, a pattern may contain objects from both spaces and are named using these objects' labels (possible because they are unique). These representative nodes are called structural and value nodes for type and instance representation respectively. However, only type arcs can be defined in the pattern. Since the user never controls the attribution of labels to instance arcs, it wouldn't make sense to allow him the specification of these labels. A pattern is constituted nodes and arcs. Nodes are divided into two kinds: structural and value nodes. Structural nodes represent nodes that are types and value nodes represent nodes that are instances. Only one kind of arc exists in the database because only types are represented.

A pattern graph is thus defined as graph, i.e. P = (PN, PA, PL, label, source, dest) where PN is a set of pattern nodes, PA is a set of pattern arcs, and a set of labels PL $\subseteq$ (TL $\cup$ IL).

Two kinds of pattern nodes exist: structural and value nodes. A structural node psn $\in$ PSN is identified by its label psnl $\in$

(PSNL $\subseteq$ TL). A value node pvn $\in$ PVN is identified by its label pvnl $\in$ (PVNL $\subseteq$ IL). PNL = PSNL $\cup$ PVNL, PN = PSN $\cup$ PVN. A pattern arc pa $\in$ PA is defined by (psl, pal, pdl), where psl, pdl $\in$ PSNL $\cup$ PVNL, pal $\in$ (PAL $\subseteq$ TAL). psl is called the source of the arc, and pdl is called the destination. We also have implicitly PNL $\cap$ PAL = $\varnothing$ and PN $\cap$ PA = $\varnothing$. We define three operations: label(o): o $\in$ (PN $\cup$ PA) $\rightarrow$ PL, which returns the label of an object o $\in$ (PN $\cup$ PA); source(pa): pa $\in$ PA $\rightarrow$ pnl $\in$ PNL which returns the label source for the arc pa $\in$ PA; dest(pa): pa $\in$ PA $\rightarrow$ pnl $\in$ PNL which returns the label destination for the arc pa.

Patterns may contain objects carrying labels that objects included in other patterns already carry. This is possible because all patterns are independent from the others and do not form a single space.

We cannot allow the definition of more than one node baring a given name in a given pattern. This is due to the way arcs are defined and the kind of object identity supported by our model. However, we need to be able to express queries where the same concept may be used in different contexts at the same time because two different instances are considered. Moreover, the possibility of cycles makes the writing of patterns uneasy if nodes cannot be repeated. We allow the definition of many nodes and arcs baring the same name thanks to the definition of aliases. We simply define an alias dictionary which is consulted whenever an element cannot be found in an operation or an invalid type encountered. These labels must still be unique and they provide a way to keep the uniqueness of labels in graphs. We could have modelled aliases as synonyms in the same way we modelled synonyms for types and for instances. This would have produced a graph aspect at a meta-level and a more uniform system, but would have generated a much more complicated structure for the database (thus more complicated for the user). We have to keep in mind that the system must be usable by non-computing people. Later, we do not include aliases in diagrams or formulas for simplification. Semantically we give many different virtual labels to a single element in the database, and each time one of these virtual labels is used in operations such as type or conformity checking, the virtual label is resolved to the real label.

Type unions and instance synonyms are voluntarily excluded from the definition of patterns because they do not represent real word objects and only those objects can be queried. Moreover, there is not necessarily structural uniformity in the types participating in the definition of the union so querying their structure is uneasy. Indeed, if we allow the use of synonyms in the querying, patterns would become far more complicated because they would have to take in count the each possible structure resulting of the exploration of the synonymy group. Moreover, we assume that when a user queries the database, he/she already has a clear idea of what is to be queried, therefore knows what the queried structure is. For example, although we have defined a taxonomic type union type previously, if a user queries a structure where it is involved, he/she knows if the result must be a Specimen of a NT concept because their meaning is different.

For a pattern to be valid, it has to represent a valid subgraph of the schema, possibly through instances. It means that in order to check whether a pattern is valid, it is necessary to find the schema which it represents (by following references from patterns to types and instances using labels, and then possibly from instances to types). The elements found this way must be conform to the schema on which the pattern is applied. In order to check that a structural node is conform to the schema, we first find the type that carries the same label as the structural node, and the type of all arcs that originate from the structural node. Then as we did for instance type conformation, we check that the relationships at type level are the same than in the pattern. We proceed in a similar way for value nodes, except that we first start by finding the instance which label is the same as the value node, then its type, and only then we can check that the pattern node conforms to the type. The conformation checking for an arc is made harder by the presence of types and instances at the same time (i.e. the origin and the destination of the arc may be either a representation of a type or a representation of an instance). Therefore, for each node at an end of the arc, we must search for an instance then a type if the node is a value node or a type if it is a structural node. Then when the type carrying the same label as the pattern arc is found, it is possible to check whether it follows the definition of that type. When all components of the pattern conform to the definition of the schema, the pattern is said valid.

It is obvious that our patterns can only define equality and the presence of a conjunction of objects in the instance graph, like it is the case in the first language defined for GOOD. This limitation can be seen as too restrictive for some uses of the database system even though it is often enough to express many queries. We can extend this definition of a pattern to incorporate the notion of negativity. A negation can be a negative arc or a negative node. In the definition of type conformity, a negative node or a negative arc does not make any difference. Indeed, we only allow the querying according to the schema of the database. The non-existence of a node or an arc is checked on instance values, not on types. In consequence, it is not possible to query objects that are not of a given form/type. The effect of the presence of negative elements is only visible on matching (see below). We thus define negatives arcs and nodes like positive arcs and nodes but mark them as negative, i.e. we specialise the definitions of pattern value nodes into positive and negative nodes. We define in the same way the set of pattern positive value nodes and the set of pattern negative value nodes, PPVN and PNVN respectively with $PVN = PPVN \cup PNVN$, and $PPVN \cap PNVN = \emptyset$. We proceed in a similar way to define pattern positive and negative arcs sets, PPA and PNA respectively with $PA = PPA \cup PNA$, and $PPA \cap PNA = \emptyset$. The meaning of negativity for arcs and nodes is different. For value nodes, negativity means “any instance of the same type that has a different label”, whereas for types it means “not one instance of this type between the two nodes”. We also restrict the possibilities expressed by our patterns: a negative arc can only be created between two positive nodes. Indeed, if the origin if an arc is positive, the arc itself is negative and the destination of the arc is negative, the query is meaningless. It would be asking for a configuration where there is no relationship between a node and not another node which value is not specified. Two value nodes representing the same instance node can coexist in the pattern if one if positive and the other is negative. Indeed, we include the negation in the label of the node in order to form a node with a new label, but representing the same instance in the database.

4.2. Matching

After the definition of a pattern, the matching may be required exact, or approximate. An exact matching is a matching where the pattern reflects exactly the structure searched in the database, whereas an approximate matching involved the definition of paths, regular expressions, or variable sections of the pattern.

4.2.1. Exact matching.

Once a pattern is defined, it is necessary to match it against the database in order to find the element configurations that fulfil the user requirements. There are many ways of performing pattern matching on graphs and especially on trees (a tree is a connected graph where any two nodes are connected by a unique chain). These techniques are very useful for pattern matching in biology in order to discover molecules or DNA sequences (e.g. [Jona97]). Some of these algorithms are 2D algorithms (e.g. [B-YR93], [KU94]) because tree representations of strings where sequences are searched allow an easier finding of incomplete sequences (e.g. [ZSW94]), or because the text searched has a two-dimensional structure. Since 2D matching is a complex problem, one approach is the reduction of 2D structures to 1D structures. One possibility is to attempt a match by columns (or rows) using a simple 1D matching technique (e.g. Knuth-Morris-Pratt [KMP77]). Another possibility is the transformation of the 2D text into a 1D text by representing repeated patterns by a new character and so constructing a string representation of the matrix on which an automaton or the KMP algorithm can be run. In that case the final string matching is fast, but the pre-processing time is expensive. Very few pure 2D algorithms exist. One common idea is the use of periodicity in the pattern in order to reduce the number of matches attempted. Indeed, if there is no point where the pattern overlaps with itself other than the origin, the number of potential locations of matches is greatly reduced (we know that the next match has to be outside the zone currently covered by the pattern). A similar idea is used by [B-YR93] for reducing the search space.

The main obstacle to the implementation of these algorithms is the time and the extra-space required. These algorithms may adopt a positive approach (which mean that they try to find where the matching can occur) or a negative approach (which means that they try to find where a matching cannot occur) depending on the likelihood of a match in order to speed-up the algorithm. Although these algorithms work on any matrix and any pattern with a given alphabet, they cannot cope with one of our problems: the matrix representing the graph may be in an unexpected configuration. This is due to the fact that columns and rows in a matrix can be swapped without changing the meaning of the matrix. Moreover, these algorithms often use tree structures whereas our structure is a graph, therefore without unique root and unique path between any two nodes. It is then clear that we have to proceed to the matching by cutting down the element searched into smaller entities that can be found without being disrupted by unnecessary data.

Graph systems such as Hyperlog have a different approach to the pattern matching. In this model, queries are attached to a type. Therefore, a pattern must also be defined relatively to a type. This simplifies the search to the matching of similar configurations of nodes by enumerating attributes. However, this approach may be restrictive if patterns involving very complex structures are necessary. Moreover, we have chosen not to attach patterns to certain types (see pattern definition) thus our approach is different.

The first step of our work on pattern matching consists in finding a good representation of our graphs. This representation needs to capture two aspects of our graphs: nodes and edges, and their relationships. A possibility is the extraction of path strings that would hold all information about a path in the graph, then match them with classical 1D matching techniques (e.g. [Wat97]). However, this technique seems to require a very time and space consuming pre-processing of the graph in order to extract all possible paths. Even if the graph was traversed dynamically while matching attempts occur, the presence of cycles and loop make a matching hard to detect (these algorithms are designed for trees, not general graphs). The more sensible approach is guided by what a type means: a node conforms to its type if it is involved in the same relationships as its type. Since patterns are derived from types, matching a pattern is almost finding the type of an unknown pattern. Moreover, our model is centred on relationships, so the pattern matching is a matching of relationships. Therefore, the matching of a pattern is divided into three phases:

- extraction of the relationships that compose the pattern
- extraction of the relationships that exist in the instance space
- matching of the relationships

The set of relationships in which a node (pattern or instance) is involved is divided into two subsets: incoming relationships and outgoing relationships. This distinction is necessary because incoming relationships are manly the presence of other objects in the system, and thus are not controllable. Each set contains the description of relationships in the following form: {source node, arc, destination node}. If the set currently considered is the incoming set of the node, it must appear in all destination node slots (respectively it must appear in the source node if the set considered is the outgoing set). Once the sets of relationships are defined for the pattern to be matched, we proceed similarly for instances. We can optimise the search by reducing the search space to the types and instances we are interested in. For example, if a positive structural pattern node is present in the pattern, only instances belonging to the type represented by the pattern node will be considered. If a negative value node is present, only instances belonging to the same type but with different labels will be considered in this part of the search.

Then, we proceed to the matching of the pattern relationships and of the instance relationships. For each set of relationships related to a node in the pattern, we only keep the instances that satisfy all of them. If no relationship exists in a set, then any number of relationships of any nature is valid as a match. If a negative arc is involved, we match the positive form of the relationship and consider any matching a wrong answer that we ignore.

After the previous step, we might have extracted instances that satisfy a part of the pattern, but do not interact with any other instance extracted from the pattern. For example, when an NT and its Epithet must be extracted with a condition on the Epithet, some matches may occur that do not satisfy the condition put on the Epithet. We remove these matches by repeatedly removing instances that are involved in relationships (selected relationships according to the pattern) which reference nodes not selected by the matching. We stop the process until the result set no longer changes.

Example:
We imagine a database defining the following schema (figure 2):

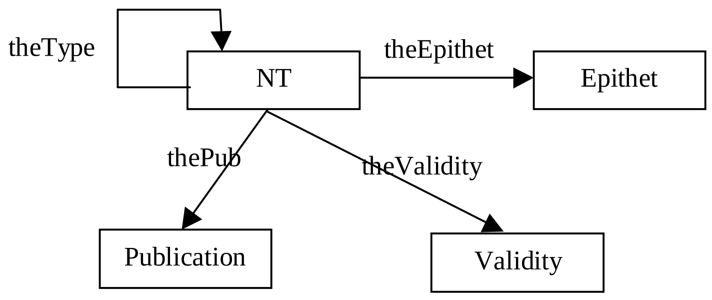


Figure 2: Sample schema

For simplicity, let us imagine a database containing the data (instances) shown in figure 3.

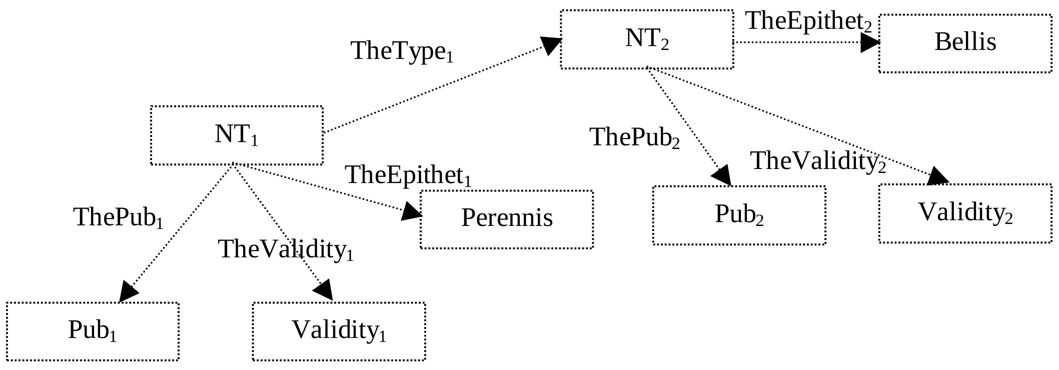


Figure 3: Sample data

In this diagram, instances are represented dotted lines.

If for example we want to find the publications that are associated to an NT which epithet is Perennis, we can define the following pattern (figure 4):

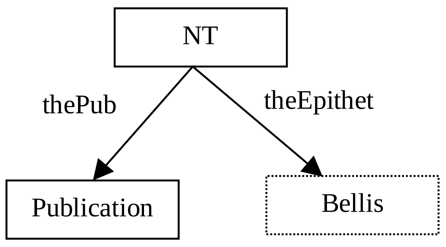


Figure 4: Sample pattern

In this diagram, pattern elements representing types are represented by solid lines, and those representing instances by dotted lines. If negative arcs or nodes are to be represented, they are assigned bold lines graphics.

The matching proceeds like follows:

1. Extract the pattern relationships

NT	in = {}
	out = {{NT, thePub, Publication}, {NT, theEpithet, Bellis}}
Publication	in = {{NT, thePub, Publication}}
	out = {}
Bellis	in = {{NT, theEpithet, Bellis}}
	out = {}

Extract the interesting instance relationships from the instance graph

NT_1	in = {}
	out = {{NT_1, $theType_1$, NT_2}, {NT_1, $theValidity_1$, $Validity_1$}, {NT_1, $thePub_1$, Pub_1}, {NT_1, $theEpithet_1$, Perennis}}
NT_2	in = {{NT_1, $theType_1$, NT_2}}
	out = {{NT_2, $theValidity_2$, $Validity_2$}, {NT_2, $thePub_2$, Pub_2}, {NT_2, $theEpithet_2$, Bellis}}
$Validity_1$	in = {{NT_1, $theValidity_1$, $Validity_1$}}
	out = {}
$Validity_2$	in = {{NT_2, $theValidity_2$, $Validity_2$}}
	out = {}
Pub_1	in = {{NT_1, $thePub_1$, Pub_1}}
	out = {}
Pub_2	in = {{NT_2, $thePub_2$, Pub_2}}
	out = {}
Bellis	in = {{NT_2, $theEpithet_2$, Bellis}}
	out = {}
Perennis	in = {{NT_1, $theEpithet_1$, Perennis}}
	out = {}

2. The resulting set of relationships is the following after removing unnecessary matches:

NT_2	in = {}
	out = {{NT_2, $thePub_2$, Pub_2}, {NT_2, $theEpithet_2$, Bellis}}
Pub_2	in = {{NT_2, $thePub_2$, Pub_2}}
	out = {}
Bellis	in = {{NT_2, $theEpithet_2$, Bellis}}
	out = {}

The result of the pattern matching is in consequence the following instance graph:

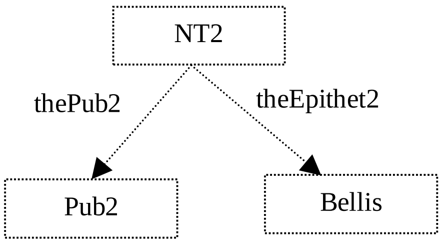


Figure 5: Result of the pattern matching

In some cases, many matches are possible in the database according to a given pattern. In this case, each component (sub connected graph) of the resulting graph is a match for the pattern. Components are simply found by going through the matrix and following arcs. Each time a node is found, the algorithm checks if it already belongs to one of the found components, if not it creates a new set (component).

It appears clear now that Prometheus achieves recursive behaviour by extracting whole subgraphs from the database. Since our structure is a graph containing possibly loops and cycles, a graph traversing approach would be much more complicated and inefficient. For example [BDS95] had to introduce the complex concept of tree markers in order to deal with cycles. We do not think that this approach is suitable for people with not computing background and is not user friendly [Cluet97].

4.2.2. Approximate matching

We have not yet developed an efficient algorithm for incomplete patterns or regular expressions. Some techniques exist in the literature for 1 dimensional spaces and patterns (e.g. [AAL97], [A-YP96]), or for 2D approximate matching (e.g. [ZSW94]). One expensive possibility is to consider each relationship between two nodes as being a circuit (succession of arcs, possibly cyclic) and then find ways through the graph. The problem of this method is its cost in term of time: each relationship defined in the pattern implies a circuit search through the graph. The number of circuits in a graph may be enormous and an exhaustive search uneasy because of the number of possibilities created by cycles.

A possible approximate matching is the matching of paths using a graph traversing approach on a particular arc (the equivalent of a form of simple path expressions). For example, the path between the rank family and the rank species can be of any length, but always follows the relationship NextRank between Rank nodes. This is an unusual kind of queries that cannot be represented be the languages we know on graphs, but they might be required in order to implement consistency rules in the system (see later).

Like it was done in the case of exact matching, we can consider approximate matching with swaps and apply it either on a 1D representation of the 2D space, or repeatedly on the 2D matrix taking great care of keeping the constancy of the pattern (e.g. if two elements are swapped, the whole column or row must be swapped in the pattern before proceeding to the match). The feasibility and efficiency of this idea have not been tested yet. Moreover, the differences that may occur obey certain rules. For example, the insertions must represent an existing path in the database. Like for the exact matching, we hope to find a practical and efficient way to extract the matches from the graph because first the alphabet is very limited and because we can use the type information available to us.

The matrix we used has the property to show paths of length n when it is multiplied n–1 times by itself. For example M^2 will show all paths of length 2 between every pair of nodes. This can be used in order to find out the paths following a type arc in a repeated fashion.

4.3. Extraction of information

It must be possible to represent in a query the structure that must be matched against the database (the pattern) but also the structure returned (from a user point of view but also for applying the operations described above). For example, a pattern may contain the definition of a Person concept, but only the name associated with it is required as an answer. Thus we define a query as being constituted of two parts, the head and the body. The body contains one or more patterns that will be matched against the database, and the head contains the structure that is required (a single pattern). This structure returned must only contain nodes and edges defined in one of the patterns constituting the body and are only labelled with type names. It is the equivalent of the SELECT clause in an SQL statement. This pattern must be directly derived from the schema in order to define the structure of the data returned. Therefore, it can be extended to the schema, i.e. if values are to be matched in the body of the query, only their type can be included in the head. For example, if author Mark is part of the body of the query and is to be included in the resulting data, the type Author must be included in the head of the query.

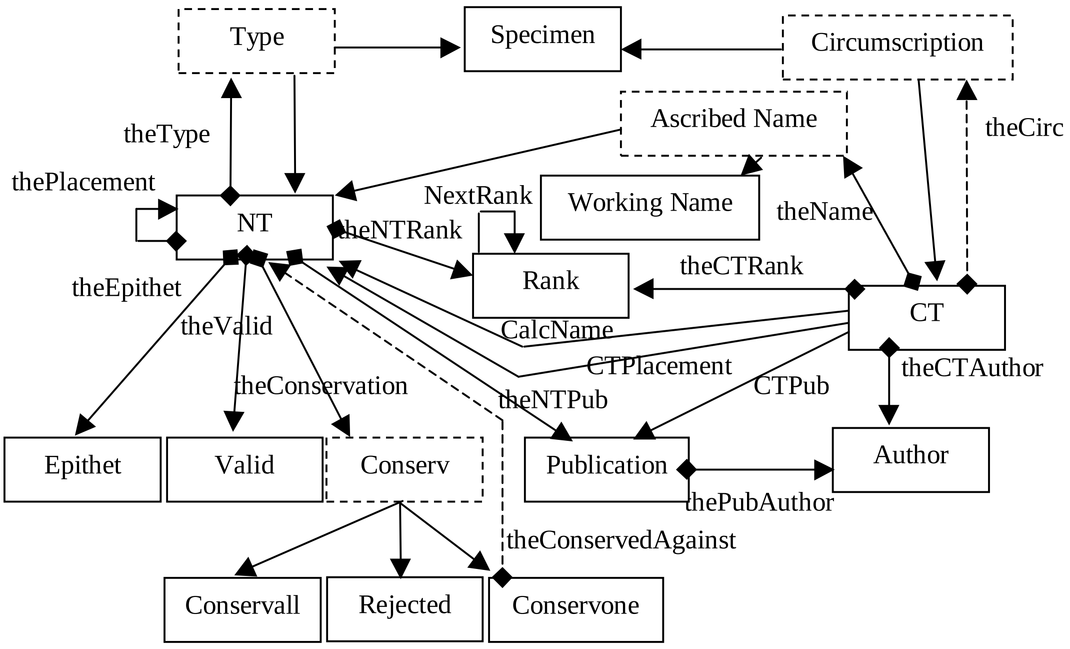
In the above example, the head of the query is simply the set of nodes Publication, which as a result of the query run against the database only contains Pub_2.

4.4. Programs.

A program consists of a series of queries for information retrieval, and a set of operations applied on the results of the queries.

5. Database Example

Using the concepts defined in our model, we can redesign the taxonomic model we have developed in [PK99]:



In this diagram types are represented by solid boxes and union types by dashed boxes. Single arcs are represented by a solid line and multi-valued arc by dashed lines. Aggregation arcs are represented with a starting diamond whereas associations are not. Note that arcs linking union type nodes to other nodes are not names. This is because their name is not attributed by the user.

This schema differs from the object-oriented one by the absence of inheritance (replaced by union types), and the fact that Publication is not specialised into OriginalPublication. Indeed, a publication is made original by its relationship to an NT (Nomenclatural Taxon), not by its nature.

Since types are unique, the choice of names is not always easy. By tradition, we use "the" as a prefix for arcs representing aggregation because of the implicit meaning that the targeted node is special (part of the origin node). A good way to manage arc names could be to include the source node name and the destination node name in their names, although it produces very long names. In any case, one must be careful in the choice of names because the creation of a type node with a particular name implies that no instance node will ever have that name.

Now, we can imagine that we want to populate this model with a very small simplified classification containing only information about Bellis Perennis. First, we enter the required information about names (NT):

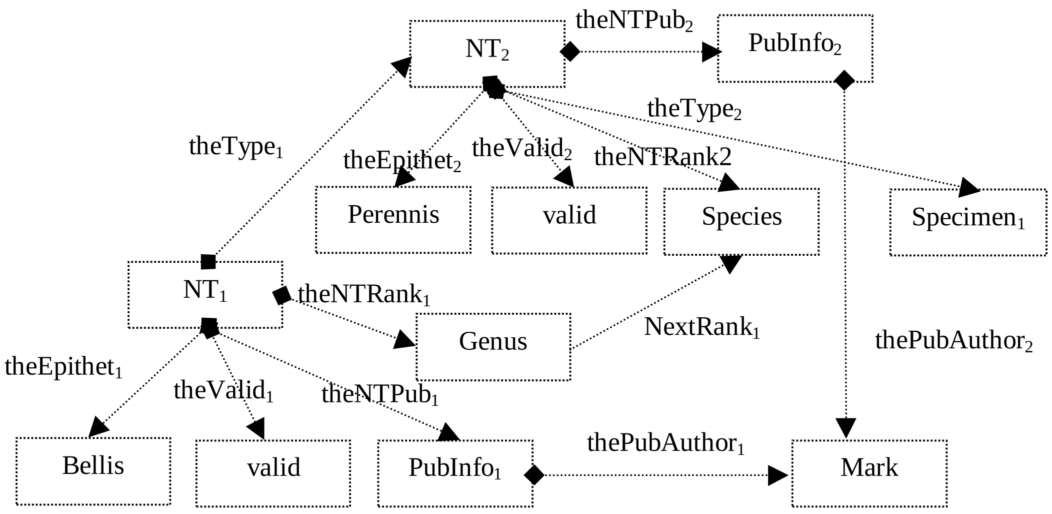


Figure 7: Instance graph for names

Note that in this diagram, the atomic node valid has been repeated although it exists only once in the database (uniqueness of labels) in order to clarify the display.

This instance diagram shows how the structure is repeated at different levels (a level being identified by its rank object) to form a recursive structure. It shows also that we are not in presence of a tree, but a directed graph (note for example that the Mark node is reached by at least two different circuits), which is due to the sharing of data between different levels of the hierarchy and different hierarchies.

We can associate the following set of relationships with this instance graph.

NT_1	in = {}
	out = {{NT_1, $theEpithet_1$, Bellis}, {NT_1, $theValid_1$, valid}, {NT_1, $theNTPub_1$, $PubInfo_1$}, {NT_1, $theNTRank_1$, Genus}, {NT_1, $theType_1$, NT_2}}
Bellis	in = {{NT_1, $theEpithet_1$, Bellis}}
	out = {}

valid	in = {{NT_1, $theValid_1$, valid}}
	out = {}
PubInfo1	in = {{NT_1, $theNTPub_1$, $PubInfo_1$}}
	out = {{$PubInfo_1$, $thePubAuthor_1$, Mark}}
Genus	in = {{NT_1, $theNTRank_1$, Genus}}
	out = {{Genus, $NextRank_1$, Species}}
Mark	in = {{$PubInfo_1$, $thePubAuthor_1$, Mark}, {$PubInfo_2$, $thePubAuthor_2$, Mark}}
	out = {}
NT_2	in = {{NT_1, $theType_1$, NT_2}}
	out = {{NT_2, $theEpithet_2$, Perennis}, {NT_2, $theValid_2$, valid}, {NT_2, $theNTPub_2$, $PubInfo_2$}, {NT_2, $theNTRank_2$, Species}, {NT_2, $theType_2$, $Specimen_1$}}
Perennis	in = {{NT_2, $theEpithet_2$, Perennis}}
	out = {}
$PubInfo_2$	in = {{NT_2, $theNTPub_2$, $PubInfo_2$}}
	out = {{$PubInfo_2$, $thePubAuthor_2$, Mark}}
Specimen	in = {{NT_2, $theType_2$, Specimen}}
	out = {}
Species	in = {{Genus, $NextRank_1$, Species}, {NT_2, $theNTRank_2$, Species}}
	out = {}

Then, we can enter data related to the classification of specimens (CT side) as follows:

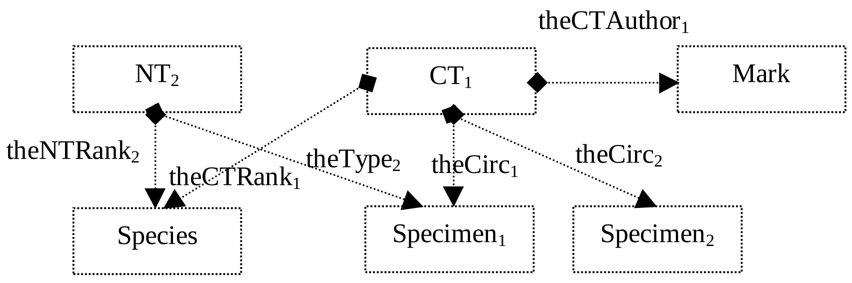


Figure 8: Instance graph for classification

In this diagram, only the NT2 and the Species nodes have been copied from figure 8 to show how these diagrams are related.

The set of relationships associated with this part of the instance graph is as follows:

NT_2	in = {}
	out = {{NT_2, $theNTRank_2$, Species}, {NT_2, $theType_2$, $Specimen_1$}}
Species	in = {{NT_2, $theNTRank_2$, Species}, {CT_1, $theCTRank_1$, Species}}
	out = {}
$Specimen_1$	in = {{NT_2, $theType_2$, $Specimen_1$}, {CT_1, $theCirc_1$, $Specimen_1$}}
	out = {}
$Specimen_2$	in = {{CT_1, $theCirc_2$, $Specimen_2$}}
	out = {}
Mark	in = {{CT_1, $theCTAuthor_1$, Mark}}
	out = {}
CT_1	in = {}
	out = {{CT_1, $theCTRank_1$, Species}, {CT_1, $theCirc_1$, $Specimen_1$}, {CT_1, $theCirc_2$, $Specimen_2$}, {CT_1, $theCTAuthor_1$, Mark}}

As a first query, we can ask for the extraction of all NTs and their Epithet. The query has two parts: the pattern that will match data, and the head, that specifies what data are actually extracted from the match. In our present case, the head consists of the NT, its Epithet, and the arc joining them. The pattern, derived from the schema, is as follows (figure 9):

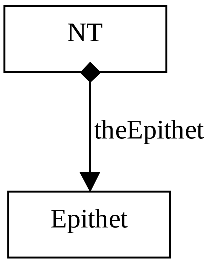


Figure 9: Pattern for query 1

The corresponding set of relationships is as follows:

NT	in = {}
	out = {{NT, theEpithet, Epithet}}
Epithet	in = {{NT, theEpithet, Epithet}}
	out = {}

The first step in the matching of the pattern consists in keeping only instances that are interesting for us. We see that we can keep only NT and Epithet instances and their related relationships:

NT_1	in = {}
	out = {{NT_1, $theEpithet_1$, Bellis}}
NT_2	in = {}
	out = {{NT_2, $theEpithet_2$, Perennis} }}
Perennis	in = {{NT_2, $theEpithet_2$, Perennis}}
	out = {}
Bellis	in = {{NT_1, $theEpithet_1$, Bellis}}
	out = {}

We don't need to apply the next steps of the algorithm because the matrix is already consistent and would not change. When we extract the connected subgraphs, we see that the answer to the query consists of two graphs (figure 10):

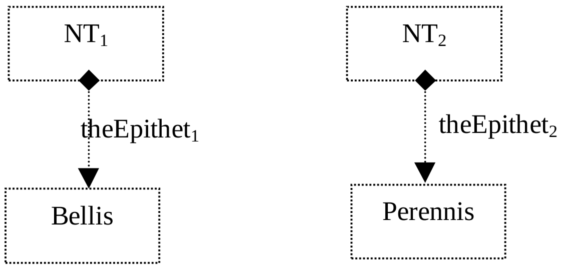


Figure 10: Result of the first query

As a second example, we can write a recursive query. We can ask for all NT with their Epithet, which have been published by author Mark, and structured as a hierarchy. In a recursive query language, answering this query would mean traversing the instance graph until a configuration is found. Since we do not use a recursive query language per se, our approach is the description of the recursive link as it is part of the model. The corresponding pattern is the following (figure 11):

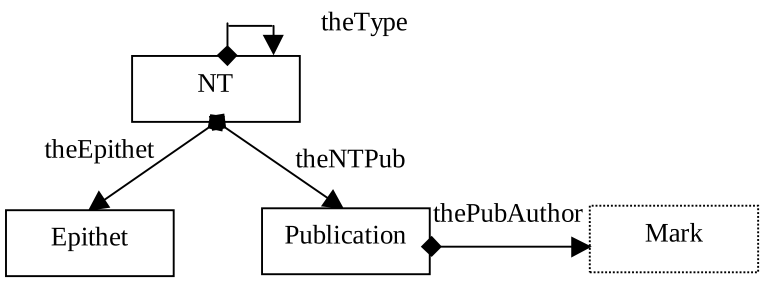


Figure 11: Pattern for query 2

The corresponding set of relationships is:

NT	in = {{NT, theType, NT}}
	out = {{NT, theType, NT}}
Epithet	in = {{NT, theEpithet, Epither}}
	out = {}
Publication	in = {{NT, theNTPub, Publication}}
	out = {{Publication, thePubAuthor, Mark}}
Mark	in = {{Publication, thePubAuthor, Mark}}
	out = {}

The application of this pattern to the instances set of relationships:

NT_1	in = {}
	out = {{NT_1, $theEpithet_1$, Bellis}, {NT_1, $theNTPub_1$, $PubInfo_1$}, {NT_1, $theNTRank_1$}, {NT_1, $theType_1$, NT_2}}
Bellis	in = {{NT_1, $theEpithet_1$, Bellis}}
	out = {}
$PubInfo_1$	in = {{NT_1, $theNTPub_1$, $PubInfo_1$}}
	out = {{$PubInfo_1$, $thePubAuthor_1$, Mark}}
Genus	in = {{NT_1, $theNTRank_1$, Genus}}
	out = {}
Mark	in = {{$PubInfo_1$, $thePubAuthor_1$, Mark}, {$PubInfo_2$, $thePubInfo_2$, Mark}}
	out = {}
NT_2	in = {{NT_1, $theType_1$, NT_2}}
	out = {{NT_2, $theEpithet_2$, Perennis}, {NT_2, $theNTPub_2$, $PubInfo_2$}}
Perennis	in = {{NT_2, $theEpithet_2$, Perennis}}
	out = {}
$PubInfo_2$	in = {{NT_2, $theNTPub_2$, $PubInfo_2$}}
	out = {{$PubInfo_2$, $thePubAuthor_2$, Mark}}

Since this matrix is connected, the resulting graph is as follows (figure 12):

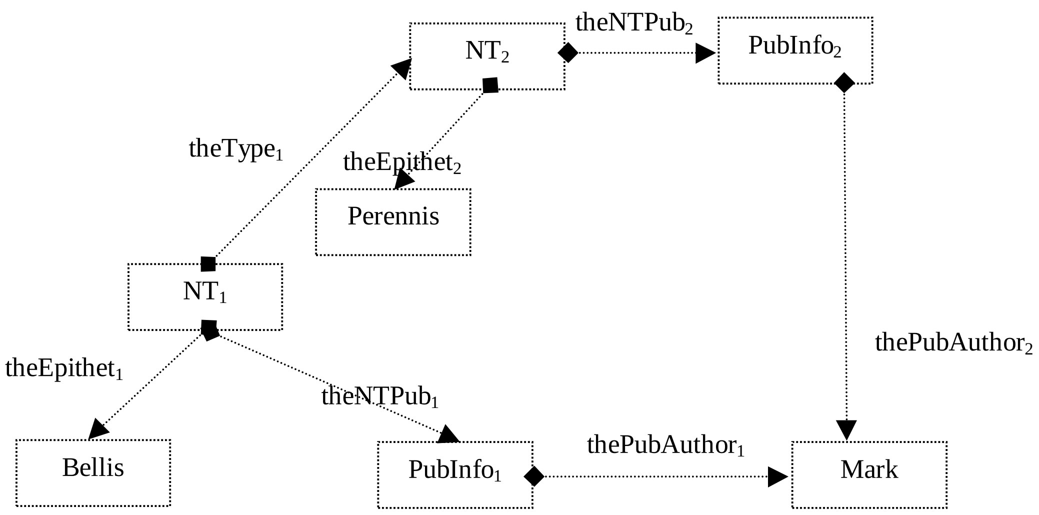


Figure 12: Solution for query 2

The last step is the extraction of the structures required by the user in the head of the query, i.e. the instances of NT, their arc to an instance of Epithet and the instances of Epithet, and the type arc joining NTs.

As a third example, we can query the database for the NT couples where one NT of rank Species is the type of another NT of rank Genus. This involves the use of patterns where nodes and arcs are repeated. The pattern is as follows (figure 13):

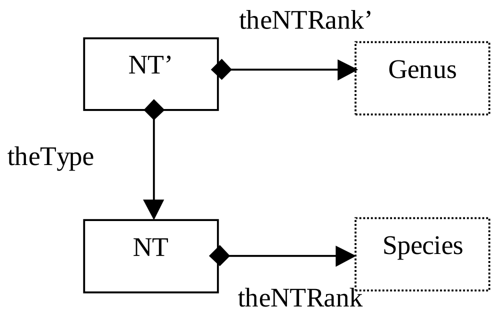


Figure 13: Third sample pattern

The set of relationships associated with this template is as follows:

NT'	in = {}
	out = {{NT', theType, NT}, {NT', theNTRank', Genus}}
NT	in = {{NT', theType, NT}}
	out = {{NT', theNTRank, Species}}
Species	in = {{NT, theNTRank, Species}}
	out = {}
Genus	in = {{NT', theNTRank', Genus}}
	out = {}

The application of this pattern to the instance graph define before results in the following set of relationships:

NT_1	in = {}
	out = {{NT_1, $theNTRank_1$, Genus}, {NT_1, $theType_1$, NT_2}}
Genus	in = {{NT_1, $theNTRank_1$, Genus}}
	out = {}
NT_2	in = {{NT_1, $theType_1$, NT_2}}
	out = {{NT_2, $theNTRank_2$, Species}}
Species	in = {{NT_2, $theNTRank_2$, Species}}
	out = {}

The resulting graph is then shown in figure 14.

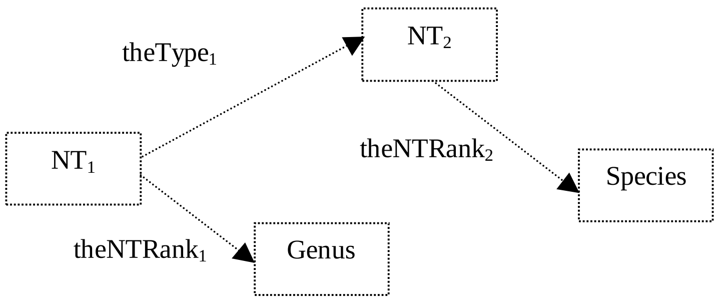


Figure 14: Result of query 3

Finally, we can query the database using a pattern containing a negative node. For example, we can ask the NT that has not been at rank Genus. The associated pattern is (figure 15):

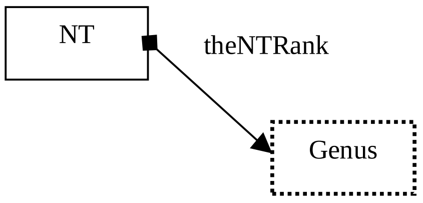


Figure 15: Pattern with a negative node

The associated set of relationships is (¬ means negation):

NT	in = {}
	out = {{NT, theNTRank, ¬Genus}}
¬Genus	in = {{NT, theNTRank, ¬Genus}}
	out = {}

The matching if this pattern on the instances produces the following result. It is important to note that Genus was removed as not relevant because it was a negative node and all nodes of the same type as Genus (but not Genus) included. Then we match any arc that reaches one of these elements (or none).

NT_1	in = {}
	out = {}
NT_2	in = {}
	out = {{NT_2, $theNTRank_2$, Species}}
Species	in = {{NT_2, $theNTRank_2$, Species}}
	out = {}

The application of the rest of the algorithm eliminates NT_1 because it does not conform to the pattern. The resulting graph is then the following (figure 16):

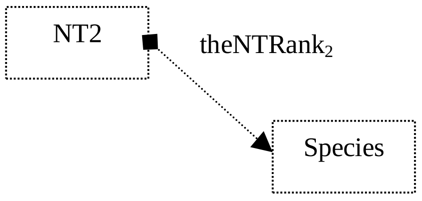


Figure 16: Resulting graph of query 4

6. Conclusion

In this document, we have defined a query language for the Prometheus database system. This query language is based on graphs and is itself composed of graphs. A query is composed of many patterns (in the body of the query) that are all matched against the database and return sets of matching data. The resulting set is then matched by another set of patterns (in the head of the query) that filter the result of the query in order to present the required kind of information to the user.

This query language has been built on top of the model presented in [RKB99]. It allows the user to query and manipulate the database and is at the moment text based. But since graphs are naturally represented graphically, a graphical user interface will be built on top of this query language.

Further work includes the definition of mechanisms for restricting the amount of data available to the user in order to make taxonomic work more easy, and the definition of integrity constraints mechanisms.

7. References

[PL94] A. Poulovassilis, M. Levene, "A nested-graph model for the representation and manipulation of complex objects", ACM Transactions on Information Systems Vol. 12 Issue 1, pp 35-68 (1994)

[PK99] M. Pullan, J. Kennedy, *et al.*, "The Prometheus Taxonomic Model ", submitted to Taxon, 1999

[RK97] P. J. Rodgers, P. J. H. King, "A graph rewriting visual language for database programming", Journal of Visual Languages & Computing Vol. 8 Issue 5/6, pp 641-674 (1997)

[RKB99] Cédric Raguenaud, Jessie Kennedy, Peter J. Barclay, “The Prometheus Database Model”, Technical report, School of Computing, Napier University (1999)