The $\mathcal{L}_{yri}\mathcal{C}$ Language: Querying Constraint Objects

Alexander Brodsky
Dept. of Information and Software Systems Engineering
George Mason University
Fairfax, Virginia, 22030-4444, USA
brodsky@isse.gmu.edu

Yoram Kornatzky
Dept. of Computer Science
University of Toronto,
6 King's College Road, Toronto, Canada
yoramk@db.toronto.edu

Abstract

Proposed in this paper is a novel data model and its language for querying object-oriented databases where objects may hold spatial, temporal or constraint data, conceptually represented by linear equality and inequality constraints. The proposed $\mathcal{L}yri\mathcal{C}$ language is designed to provide a uniform and flexible framework for diverse application realms such as (1) constraint-based design in two-, three-, or higher-dimensional space, (2) large-scale optimization and analysis, based mostly on linear programming techniques, and (3) spatial and geographic databases. $\mathcal{L}yri\mathcal{C}$ extends flat constraint query languages, especially those for linear constraint databases, to structurally complex objects. The extension is based on the object-oriented paradigm, where constraints are treated as first-class objects that are organized in classes. The query language is an extension of the language XSQL, and is built around the idea of extended path expressions. Path expressions in a query traverse nested structures in one sweep. Constraints are used in a query to filter stored constraints and to create new constraint objects.

1 Introduction

We propose the $\mathcal{L}yri\mathcal{C}$ data model and query language - a novel language for querying object-oriented databases where objects hold spatial, temporal or constraint data, conceptually represented by equality and inequality constraints. $\mathcal{L}yri\mathcal{C}$ provides an integration of the object-oriented and constraint paradigms in one unified framework. $\mathcal{L}yri\mathcal{C}$ is intended to be used in application realms such as (1) constraint-based design in two-, three-, or higher-dimensional space, (2) large-scale optimization and analysis, based mostly on linear programming techniques, and (3) spatial and geographic databases. $\mathcal{L}yri\mathcal{C}$ is based on the object-oriented paradigm [BEER89] by treating constraints as first-class objects with a logical object identity. The meaning of such objects is maintained by including the mapping from a constraint object (identity) into the infinite collection of points it represents as part of the logical model of the database [KV84, KLW90]. Constraints are organized in classes like other objects and can have attributes and methods that attach additional information to them (e.g. names of regions in a GIS), and operations to manipulate them (e.g. constraint conjunction), respectively. Constraint objects are used as attributes of other objects, where such attributes have an attached list of the variables that may be used to express constraints assigned to these attributes. These are used in order to allow for the possibility of jointly constraining different attributes when they are operated on together by constraint operations.

$\mathcal{L}yri\mathcal{C}$ extends flat constraint query languages [KKR93], especially those for linear constraint databases [BJM93], by incorporating constraints as a basic tool for describing spatio-temporal information in constraint databases. The $\mathcal{L}yri\mathcal{C}$ model treats each constraint object separately, instead of viewing each constraint tuple as a conjunction of all constraints, and a constraint relation as a disjunction of constraint tuples. This corresponds to the needs of spatio-temporal applications where information has to be viewed from multiple perspectives in a flexible way. Thus, we conjoin constraints in an object and its parts iff they share some variables, and appear together within the scope of a constraint operation in a query. Integrating constraints as another means of description of objects within an object-oriented data model is a natural requirement for supporting larger and more complex applications with constraint technology. Existing proposals for flat relational constraints databases [KKR93, BJM93], have the same problems in supporting complex spatio-temporal applications that standard relational systems have [KLW90].

The $\mathcal{L}yri\mathcal{C}$ query language is a superset of XSQL [KKS92], suggested by Kifer, Kim and Sagiv, as an extension of SQL to object-oriented databases, and is built around the idea of extended path expressions. These traverse complex nested structures by specifying paths in the database schema, while extracting parts of these for further manipulation and filtering. Constraint operations may be used in the WHERE clause of XSQL as boolean tests and in the SELECT clause to generate new constraint objects. Using constraints as higher-order variables permits flexible definition of classes in views whose set of instances is defined by means of constraint formulas, e.g. defining a subclass of geographical objects corresponding to each region described as a constraint. $\mathcal{L}yri\mathcal{C}$ provides great practical expressive power while still having PTIME evaluation data complexity. We describe a naive implementation of $\mathcal{L}yri\mathcal{C}$ through its translation into flat SQL with constraints. A future implementation will be based on a constraint algebra to be developed.

1.1 Constraints, Space and Time

Although spatio-temporal information that incorporates aspects of space and time, seems quite different from the constraint information, which is basic in applications of analysis, design and planning, these two types of information have fundamental commonality. A collection of constraints can be geometrically viewed as an object in multidimensional space, containing all points in the space that satisfy the constraints; Spatio-temporal objects or, at least, their approximations, can be described using constraints, although their physical representation can differ for efficient manipulation. Thus, constraints can serve as a unifying data type for (conceptual) representation of heterogeneous (spatio-temporal) data. In this paper we will not distinguish between constraint and spatio-temporal information. In the following, we will be referring to Constraint or Spatio-Temporal information collectively as *CST-information*, or *CST-objects*. A CST-object is thus a (possibly infinite) collection of points in a multi-dimensional space.

Using a unified constraint-based framework in $\mathcal{L}yri\mathcal{C}$ has a number of advantages. First, the uniformity of representation of spatio-temporal data and operations in terms of constraints and their manipulations avoids the need to separately build into the database system many spatio-temporal relationships (e.g. containment is expressed by implication), and predicates/operators (e.g. intersection is expressed by conjunction). Moreover, it permits an extensible collection of operations in spatio-temporal objects with a single implementation of linear constraint technology. Also, in many applications, CST-objects are more intuitively described as constraints (e.g. submarine maneuver decision aid [BVCS93]). A flexible representation of spatio-temporal relationships using linear constraints enables a combination of a number of layers of CST-objects based on different coordinate systems in the same query, by expressing with linear equalities the relationship between the coordinate systems. Linear constraint technology, including efficient algorithms for manipulating higher-dimensional constraints, can perform an order of magnitude better than *ad hoc* methods working on direct representations of CST-objects. For low-dimensional space, the best known data structures and algorithms will be used.

A fine trade-off between expressiveness and efficiency is crucial in the integration of constraint and database technologies. We believe that the domain of linear constraints that we incorporate in an object-oriented model and its query language is both *expressive* and potentially *efficient*, based on the state of art in linear constraint and computational geometry areas. We limit ourselves to linear constraints for this reason even though the data model and the $\mathcal{L}yri\mathcal{C}$ language can be generalized to any familiar kind of constraint.

There are many applications in which both conceptual representation of spatio-temporal objects and queries using constraints, and answering queries using a combination of constraint and database technologies can provide a great deal of flexibility and performance edge. We provide a brief description of two of these below. As a running example throughout the paper we would use the following office (architectural) design example.

1.2 Examples of Applications

One type of applications is design in multidimensional space. Suppose we keep a catalog of office objects such as desks, files cabinets, chairs etc. Each object has attributes such as `name`, `color`, and a spatial attribute `extent`, describing this object shape. The `extent` can be represented as a union of 3D polygons, which are described relatively to a local system of coordinates, e.g. one whose origin is located in the center of volume of the object. Since we want to reason on how office objects are to be located one with respect to the other, we capture the translation between the different coordinate systems by means of equations in the `translation` attribute.

Assume further that each desk has `drawer`. A *drawer* in turn, is characterized by its shape, or `extent`. Since a drawer can move relatively to the desk, its `extent` would be described in a local system of coordinates, probably with origin in the drawer's center of volume. To describe possible drawer's location in the desk's system of coordinates, we can keep `drawer_center` attribute which is a constraint describing a bounded line along which the center of the drawer is moving when the drawer is opening and closing. Thereby, it can be used to describe the `translation` between the coordinate systems of the drawer and the desk containing it, per each `drawer_center` location. Other office objects may have similar descriptions.

A designer then may ask queries such as: Given a room and location of a number of objects in it, can we put an additional desk such that its drawer will not touch any other object in the room, and still have an unoccupied 4×4 feet space? Can we put in a room two desks, two file cabinets and two chairs such that (1) no two objects or their opened drawers will touch each other or the walls, and (2) there will be at least 4 feet between the front of each desk and the opposite wall? Can the system give constraints describing possible interconnections of centers of objects such that the above goals are achieved? What would be the location of the above mentioned objects if we want to maximize the size of a square of available empty space? Given a collection of objects in the room, show a projection of their cut at the height of 1/2 feet. All these queries can be efficiently answered in $\mathcal{L}yri\mathcal{C}$ without using user implemented predicates or functions as will be shown below.

Somewhat similar design application, which deals with 4-dimensional space, is the submarine *Maneuver Decision Aid* (MDA) [BVCS93] currently being developed at the Naval Undersea Warfare Center. In every situation there is a large collection of goals such as "avoid land obstacle," "minimize speed," "maintain depth at 200ft," as well as many battle-management goals. Maneuvers are expressed as points in a 4-dimensional space, where the dimensions are: course, speed, depth and time. $\mathcal{L}yri\mathcal{C}$ queries can express finding the best suitable maneuver regions under interlated and possibly contradicting goals, where both maneuver regions and goals are expressed using constraints.

An interesting class of problems for $\mathcal{L}yri\mathcal{C}$ is a powerful extension of classical linear programming (LP) applications dealing with manufacturing and warehouse support such as allocation of scarce resources, scheduling production and inventory, and cutting stock. In those application a system of constraints in LP must be generalized to a database containing constraints, and the objective function is generalized to a database query containing constraints. Of course, the answer to this query may also contain constraints. Consider for example a chemical factory where different products are manufactured using a hierarchy of manufacturing processes, each described using linear constraints, and raw materials.

Using $\mathcal{L}yri\mathcal{C}$ queries one could answer questions such as: For each order of a product, what is the connection (described by constraints) among the required raw materials? Is it possible to improve the profit by 5% by buying some amount of a single raw material and then using a better manufacturing process? How much of each raw material should be purchased in order to satisfy all current orders? What are the ranges of and the connection among the quantities of all products that can be produced using the raw materials currently in stock? What is the best manufacturing process for a given set of orders? Can an order be filled only by using raw materials in inventory? etc. That type of queries involves both regular database aspects such as orders, invoicing, and ordering, whereas supplier selection, the warehousing strategy, the manufacturing process to use, and the scheduling of machines and orders on the factory floor are mathematical programming programs.

Organization of the Paper

Section 2 reviews the XSQL data model and language and is excerpted from [KKS92]. The constraint object model is presented in section 3, while section 4 describes its query language $\mathcal{L}yri\mathcal{C}$. The complexity of evaluating $\mathcal{L}yri\mathcal{C}$ queries by means of a naive implementation is discussed in section 5. Related work is discussed in section 6

2 XSQL Review

2.1 An Object-Oriented Data Model

We briefly review the object-oriented data model on which XSQL is based [KLW90, KKS92]. The description here is excerpted from [KKS92].

Objects and object identity. Objects are abstract or concrete entities in the real world, and are referred via their *logical object ids (oid)*. These are syntactic terms in the query language such as 20, $john23$, $secretary(dept77)$. We use explicit id-functions (such as $secretary$ above) to create oids. Any oid uniquely identifies an object. Oids may carry certain semantic information. For instance, we consider '20' to be the oid of the abstract object with the usual properties of the number 20.

Attributes. Objects are described via attributes, and all our objects are tuple-objects, whose fields are the values of the object's attributes. If the attribute is *scalar,* then the value is a single oid; if the attribute is *set-valued,* then the value is a set of oids.

Methods. A method, invoked in the scope of an object on a tuple of arguments, returns an answer, and, possibly, changes the state of that object (e.g., by changing the value of an attribute). As a function, each method has *arity*—the number of its arguments. An attribute is regarded as a 0-ary method.

Classes. Classes have the function of organizing objects into sets of related entities. The *instance-of* relationship between objects and classes determines which objects belong to which classes. The *IS-A* or *subclass* relationship, is defined between classes and acyclic. If a class C is a subclass of another class C', then all instances of C must also belong to C'.

Types. In object-oriented languages, the abstract values of interest are objects; types provide one of the important means of classifying objects. Another means of classification is the concept of a class discussed earlier. The type of a class is determined by the types of its methods. The type of a method in a class C is described as a *signature* of the form

$$\mathtt{Mthd} : \mathtt{Arg}_1, \ldots, \mathtt{Arg}_k \Rightarrow \mathtt{Result} \quad \textit{or} \quad \mathtt{Mthd} : \mathtt{Arg}_1, \ldots, \mathtt{Arg}_k \Rrightarrow \mathtt{Result}$$

that is attached to the definition of class C, where Arg_i and $Result$ are class names. The single arrow, $\Rightarrow$, is used in the declarations of scalar methods, while the double arrow, $\Rrightarrow$, is used for set-valued methods. The above signature is meant to say that when the method $Mthd$ is passed arguments that are instances of classes Arg_1, ... , Arg_k, respectively, the result is expected to be an instance, or a set of instances of the class $Result$, depending on whether $Mthd$ is scalar or set-valued. Note that there are actually $k+1$ (rather than k) arguments, where the 0^{th} argument is not mentioned, because it is the object of class C for which the signature is defined.

A method can have several signatures, each constraining the behavior of the method on different sets of arguments. When this is the case, the method is said to have a *polymorphic* type.

Inheritance. Methods defined in the scope of a class C are *inherited* by the subclasses of C.

Figure 1 shows an object-oriented schema of our office design example. Dashed arrows describe the IS-A hierarchy and solid arrows describe the *composition (aggregation) hierarchy.* (Attribute names that end with an asterisk denote set-valued attributes; other attributes are scalar.) For the moment, ignore the use of variables in attribute definitions and class names (to be explained in section 3). The schema describes objects in an office such as desks and file cabinets, and their parts such as drawers. Each object in the room (of class

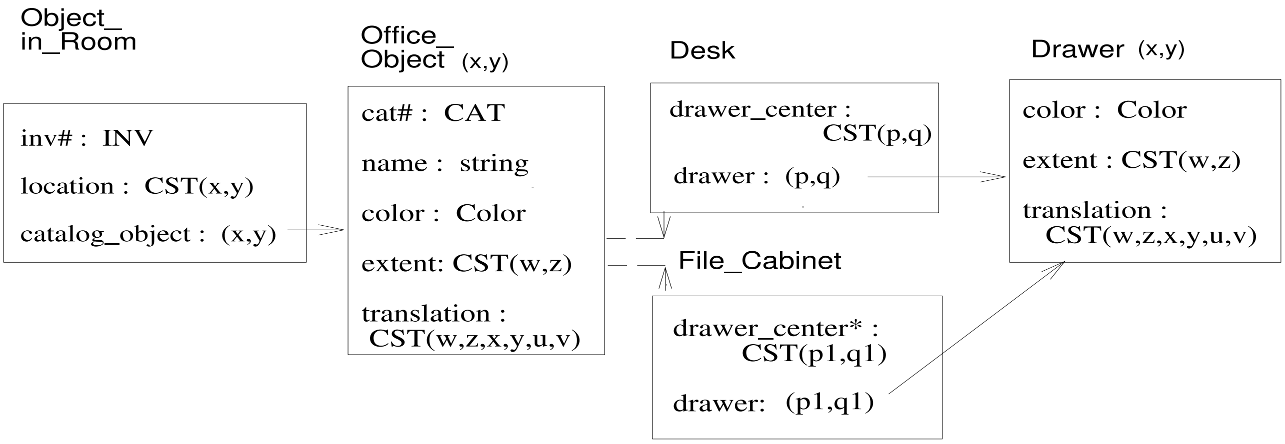

Figure 1: An Object-Oriented Database Schema

Object_in_Room) has a location in terms if the room coordinate system, and an inventory number. To simplify the example, we assume a two-dimensional world. Corresponding to such an object, there is a description of the corresponding catalog object (of class *Office_Object*) This is described in terms of a local coordinate system whose origin translation with respect to the room coordinate system is described via the `translation` attribute. The description of a catalog object includes its volume (the attribute `extent`), and for each subclass, a description of its parts. In the example, these are the drawers which are specied relative to a local coordinate system whose translation with respect to the office object coordinate system is described in the `drawer_center` attribute. The translation of the later, when the drawer is pulled in or out of the desk is specified via the `translation` attribute. We assume all drawers have the same orientation with respect to the file cabinet. Attributes of objects which range over classes of CST-objects, to be introduced in section 3 are described below.

2.2 Path Expressions

Path expressions describe paths along the composition hierarchy, and can be viewed as compositions of methods (some of them may be attributes). For example, the expression

$$\texttt{desk123.drawer.color} \qquad (1)$$

describes a path that starts in the object of class *Desk* denoted by $desk123$, continues to the drawer of $desk123$, and ends in the color of that drawer. In (1), $desk123$ is called a *selector,* and `drawer` and `color` are called *attribute expressions.*

Path expressions can be more general than the one above. Formally, a *path expression* is of the form

$$\texttt{selector}_0.\texttt{AttEx}_1\{[\texttt{selector}_1]\}.\ \cdots\ .\texttt{AttEx}_m\{[\texttt{selector}_m]\} \qquad (2)$$

where $m \geq 0$, and braces denote optional terms (i.e., only the first selector is mandatory). A selector is either *ground* (abbr. *g-selector*) or *variable* (abbr. *v-selector*). A g-selector is just an object id, and a v-selector is an *individual variable* that ranges over id's of individual objects. The attribute expressions $AttEx_1, \ldots, AttEx_m$ in (2) are either attribute names or *attribute variables* that range over attribute names. (We usually omit the classifiers, "individual" or "attribute", of variables when they are clear from the context.) Note that "higher-order" variables do not make the underlying logic second-order (see [KV84, KLW90]). Also note that any selector is also a (trivial) path; this follows from the above definition when $m = 0$. Higher-order variables over attribute and class names enable querying the database without full knowledge of its schema and are used to query and manipulate the schema. We omit a description of these here, and will provide an example of their usage in $\mathcal{L}yri\mathcal{C}$ below.

The formal definition of the meaning of a path expression requires several concepts which will be defined next. A *database path* (or just *path* when confusion does not arise) is any finite sequence of database objects, $o_0, o_1, \ldots, o_n$ ($n \geq 0$); the object o_0 is the *head* of the path and o_n is called its *tail.* A *ground instance* of a path expression is obtained by substituting an object id for each v-selector, and an attribute name for each attribute variable. Formally, a path expression E describes a set consisting of all database paths p, such that p *satisfies* some ground instance of E. A path $o_0, o_1, \ldots, o_m$, where the o_i's are objects, *satisfies* the ground instance $sel_0.attr_1\{[sel_1]\}. \cdots .attr_m\{[sel_m]\}$, if all of the following hold: (a) $o_0 = sel_0$; (b) for every $j = 1, \ldots, m$, if the selector sel_j is specified in the above path expression (recall that these selectors are optional, by definition) then $o_j = sel_j$; (c) for all $i = 1, \ldots, m$, the attribute $attr_i$ must be defined on o_{i-1}. Furthermore, if $attr_i$ is scalar, then o_i must equal the value of $attr_i$ on object o_{i-1}; if $attr_i$ is set-valued then o_i must *belong to* the value of $attr_i$ on o_{i-1}.

The set of database paths satisfying ground instances of the path expression E could be empty. This may happen because of a type error or because the path expression describes an empty set of paths in the current state of the database. For example, if E is the path expression (1) and $desk123$ is not an object of the database, then the set of paths described by E is empty. In this abstract we do not discuss typing and type errors in XSQL queries (see [KKS92] for details).

Since the path expression (1) is ground (i.e., has no variables), its last attribute is scalar, it may satisfied by at most one database path. In comparison, the path expression,

```
file_cabinet_db.drawer.color
```

would normally be satisfied on database paths that begin with the *File_Cabinet* object *file_cabinet_db* , pass through one of its drawers, and end in the object representing the color of this drawer. If *file_cabinet_db* had several drawers, then there will be several such database paths.

An expression similar to (1), can be utilized in the following query:

```
SELECT Y
FROM Desk X
WHERE X.drawer[Y].color['red']
```

Now we should consider all ground instances of the path expression in the WHERE clause. For each ground instance $x.drawer[y].color[\text{'}red\text{'}]$, we should first check *consistency* with the FROM clause; in this case, consistency means that x should be an oid of a desk. If the ground instance is consistent, then y is in the answer provided that (at least) one database path satisfies the ground instance. Observe that a path expression is used as a Boolean predicate, and a ground instance of a path expression is either true or false depending on whether it is satisfied by some database path or not.

Path expressions can be compared using the comparators $=$, $>$, etc., which compare the sets of objects in their tail (called the *value* of the expression). Comparisons can be combined using Boolean connectives (e.g., *and*). In addition, since path expressions are evaluated to sets they can be compared using such standard set-comparators such as *contains*.

The formal semantics of a query Q is defined as follows: all substitutions of oid's for variables are considered. For each substitution that is consistent with the *FROM* clause, all ground path expressions are evaluated. Next, the *WHERE* clause is evaluated. If the *WHERE* clause evaluates to *true,* then the scalar ground path expressions in the *SELECT* clause are evaluated. The result of this evaluation is a tuple of oid's that is added to the answer of the query.

Instead of merely viewing the result of a query as an ordinary relation, we can also view tuples produced by queries as new objects:

```
SELECT name=X.name, drawer=W
FROM Office_Object X
OID FUNCTION OF X,W
WHERE X.drawer[W]
```

This query has two new features. First, the *SELECT* clause gives explicit names to attributes of the output relation. Second, the *OID FUNCTION OF* clause determines an object id for each tuple in the result. A tuple of the result is generated from a pair of object id's, say x and w, that are assigned to variables X and W, respectively, and its object identity is a function of x and w, $f(x, w)$, produced by some *id-function* f [KLW90].

XSQL can be used to define views which are new classes containing oids generated by the query through an oid function. For example, to find all pairs of objects in the example schema which occupy the same volume of space due to a wrong design, we define the view[1]:

```
CREATE VIEW Overlap AS SUBCLASS OF Object
SELECT first = X, second = Y
SIGNATURE first ⇒⇒ Office_Object, second ⇒⇒ Office_Object
FROM Office_Object X, Office_Object Y
OID FUNCTION OF X,Y
WHERE X.extent[U] and Y.extent[V] and (U overlap V)
```

3 A Constraint Object Data Model

The object-oriented data model treats any kind of object such as an integer or a string as a logical object identity which might have an associated semantic meaning if it belongs to a particular class. Thus, the object '2' is identified with the usual integer 2 due to being an instance of the integer class which has the required methods such as addition and subtraction. To seamlessly integrate constraints into the data model, we view them as another kind of logical object identity, similarly to the way we view oids representing attributes and methods. The semantics is defined via a mapping, that is part of the model theory of the data model, from logical oids to infinite collections of points which represent the appropriate CST object. Thus, the semantics of CST objects which are higher-order objects, is defined based on the idea of general structures [End72], as in the whole family of F-logic languages. CST objects are organized into CST classes according to their dimension. The CST superclasses define polymorphic operations on CST objects. These are the familiar constraint manipulations such as intersection and union that can be used in logical formulas on constraints, Subclasses of these define additional attributes and methods on these objects to be used in particular applications. We next supply the definitions of linear constraints and canonical forms needed in the sequel.

3.1 Linear Constraints and Canonical Forms

We need to carefully construct the family of constraints allowed to represent CST objects, and the operators allowed in the query language, so that the data model will be closed under the language and computational costs be under control. In particular, we design the constraint domain to avoid exponential space and time explosion in terms of data complexity during constraint manipulation. To do that, we suggest four interlated families of constraints (and thus CST objects) defined formally in this subsection: *conjunctive*, *disjunctive*, *existential conjunctive*, and *disjunctive existential*. The idea is that these families will have representations as conjunction, disjunction of conjunctions, existentially quantified conjunctions, and disjunctions of existentially quantified conjunctions of linear arithmetic constraints, respectively. Moreover, we want to guarantee that for a fixed number of logical connectives, the size of the above representations and time required to achieve them be at most polynomial in the size of linear constraints. This would not be the case, for example, had we required quantifier elimination even of conjunctions of linear constraints.

In definition of the families of constraints, we introduce for convenience the *projection* logical connector. It is a variant of existential quantifier where we specify the free, rather than the quantified variables. If ϕ is a logical formula, than its *projection* on $x_1, \ldots, x_n$ is denoted

$$((x_1, \ldots, x_n) \mid \phi)$$

[1] The *overlap* predicate would be latter defined through satisfaction of constraint conjunction.

The variables $x_1, \ldots, x_n$ are called *free*. Opposed to a regular existential quantifier, they do not have to appear in ϕ, and thus a *projection* can add new free variables. The truth value of, $((x_1, \ldots, x_n) \mid \phi)$, for a given instantiation of constants into free variables, is defined recursively as the truth value of

$$(\exists \tilde{y})\ \phi$$

where $\tilde{y}$ denotes all free variables in ϕ that are not in $(x_1, \ldots, x_n)$.

A *linear arithmetic* constraint has the form, $r_1 x_1 + \cdots + r_m x_m$ *relop* r, where $r, r_1, \ldots, r_m$ are real number constants and *relop* is one of $=, <, \leq, >, \geq, \neq$.

A *conjunctive* constraint is one of the following: (a) linear arithmetic constraint; (b) if ϕ and ϕ' are conjunctive constraints, then so are $\phi \wedge \phi'$ and and a projection, $((x_1, \ldots, x_n) \mid \phi)$, where either (1) at most one, or (2) all but one of the free variables of ϕ appear in $(x_1, \ldots, x_n)$. The last operator corresponds in fact to a restricted quantifier elimination of one, or all but one variables. The idea here is that we can perform each restricted quantifier elimination in polynomial time and represent the result as a conjunction (without quantifiers) of linear arithmetic constraints.

An *existential conjunctive* constraint is one of the following: (a) a *conjunctive* constraint; (b) if ϕ and ϕ' are *existential conjunctive* constraints, then so are $\phi \wedge \phi'$, and a projection, $((x_1, \ldots, x_n) \mid \phi)$ Here, as opposed to *conjunctive* constraints, we do not have any restriction on projection (existential quantification). Clearly, any *existential conjunctive* constraint can be represented in linear time as an existentially quantified conjunction of linear arithmetic constraints.

A *disjunctive* constraint is one of the following: (a) a *conjunctive* constraint or its negation ($\neg$); (b) if ϕ and ϕ' are *disjunctive* constraints, then so are $\phi \vee \phi'$, $\phi \wedge \phi'$, and a projection $((x_1, \ldots, x_n) \mid \phi)$, where either (1) at most one, or (2) all but one of the free variables of ϕ appear in $(x_1, \ldots, x_n)$. The *projection* here, as in the case of *conjunctive* constraints, corresponds to a restricted quantifier elimination of one, or all but one variables. The idea here again is that we can perform each restricted quantifier elimination in polynomial time, and thus represent any *disjunctive* constraint as disunction of conjunctions (without quantifiers) of linear constraints.

Finally, *disjunctive existential* constraints are one of the following: (a) *disjunctive* or *existential conjunctive* constraints; (b) if ϕ and ϕ' are *disjunctive existential* constraints, then so are $\phi \vee \phi'$, and the projection $((x_1, \ldots, x_n) \mid \phi)$, where all free variables of ϕ are in $x_1, \ldots, x_n$. The last condition essentially avoids having existential quantification on a *disjunctive existential* constraint. Thus, any *disjunctive existential* constraint can be represented as a disjunction of possibly existentially quantified conjunctions of linear constraints.

According to our definitions, *existential conjunctive* and *disjunctive* constraints each include *conjunctive* constraints. *Disjunctive existential* constraints include all the others.

We briefly discuss now some computational issues related to constraints manipulation and their canonical forms, that will be used for representation of CST objects. It is important to mention that all families of constraints discussed earlier can be represented as *disjunction of possibly existentially quantified* linear constraints. We adopt here the canonical forms and their description suggested for flat constraint relations in [BJM93] to CST objects. A *canonical form* for constraints is a useful standard form of the constraints, and is generally computed by simplification and the removal of redundancy. In addition to the advantages of a standard presentation of constraints, canonical forms can provide savings of space and time.

In the class of linear arithmetic constraints, there are many plausible canonical forms. However, they can be costly to compute. Detecting redundant disjuncts is a co-NP-complete problem [Sri92], so we will perform only two simplifications of disjunctions: the deletion of each inconsistent disjunct and the deletion of syntactic duplicates. Similarly, while it is theoretically possible to eliminate all existential quantifiers from *existential conjunctive* constraints (as required in the framework of [KKR93]), the cost of this elimination and the size of the resulting constraint can grow exponentially in the size of the original constraint. Since we expect applications with large constraints, it is unrealistic to expect that all quantifiers can be eliminated. We perform only simplifying quantifier eliminations, similar to what is done in CLP($\mathcal{R}$) [JMSY92]. The *conjunctive* constraints offer the greatest scope in choosing a canonical form and can be found in [BJM93].

To sum up, the canonical form chosen is orthogonal to the $\mathcal{L}yri\mathcal{C}$ language, and will influence the semantics of the data model and language only in the sense that we may have constraints with different canonical form which represent the same CST object. This issue will be taken below when we consider the comparison of oids of CST objects.

3.2 Constraint Objects

As in [JaL87, KKR93, BJM93]), we view constraints as another means to represent a (possibly infinite) collection of points in (n-dimensional) space. For example, a constraint such as $((x, y)|\ 2x + 3y \leq 5\ )$ can be viewed as the infinite collection of points in $\Re^2$: $\{(a_1, a_2)|\ 2a_1 + 3a_2 \leq 5\ \}$. In general, we say that a constraint of the form $((x_1, \ldots, x_k)|\phi)$ *represents* a subset of $\Re^k$, defined as a collection of all points $a_1, \ldots, a_k$ that satisfy $((x_1, \ldots, x_k)|\phi)$. Equivalently, the constraint represents the k-dimensional predicate, which is *true* on exactly those points in the collection. Thus, a constraint is a higher-order object which can be viewed either as a set (collection) of points or a predicate which is true on those points which are members of the collection. We define a k-dimensional *CST object* as a subset of $\Re^k$ representable by a *disjunctive existential* constraint $((x_1, \ldots, x_k)|\phi)$. By a slight abuse of notation, we will also refer to the k-dimensional CST object as the k-dimensional predicate represented by the constraint.

To integrate higher-order objects such as sets and predicates into object-oriented languages while maintaining a first-order semantics, we use the ideas expounded in previous work [CKW89, KLW90]. Formally, a model of languages (logics) which have a higher-order syntax and a first-order semantics is a general structure [End72]. Thus, a database is a structure in which every object in the data model, including higher-order ones such as classes, and methods, has an atomic oid representing it. Thus, variables ranging over higher-order objects range over a domain of atomic oids representing this kind of objects. The fact that such an oid represents a higher-order object is captured by including in a structure a mapping from oids to the corresponding higher-order objects. Formally, as in previous data models of its kind, a $\mathcal{L}yri\mathcal{C}$ database is represented as a structure with components for the universe of the database, class membership, subclass ordering and mappings from oids to methods (attributes), and classes. We omit the details as they appear elsewhere [KLW90], and consider only the following components required in the sequel:

$$< U,\ \prec_U,\ \in_U >$$

representing the universe of the database, the subclass ordering, and the class membership. We map oids to CST objects by including a mapping from oids to infinite collections of points:

$$\mathcal{R}\ :\ U\ \mapsto\ [\prod_{k=1}^{\infty} 2^{\ (\Re^k)}]$$

whose k-th component is used to map an oid to its use as a (possibly infinite) collection of points in k-dimensional space. This mapping is applied only to oids corresponding to constraints such that for a disjunctive existential constraint, $((x_1, \ldots, x_k)|\phi)$, $\mathcal{R}^{(m)}\ ((x_1, \ldots, x_k)|\phi)$, is the m-dimensional CST object defined by the constraint $((x_1, \ldots, x_m)|((x_1, \ldots, x_k)|\phi))$. Here, if $m > k$, all $x_{k+1}, \ldots, x_m$ are new different variable names.

Oids for CST objects, similar to those of other higher-order objects, are considered disjoint only up to canonical form simplification. Thus, we may have two oids which are equivalent semantically but which would be considered different because they do not have the same canonical form after the particular simplification algorithm is applied. Constraints may be created in queries using constraint operations. The created constraints will correspond to new oids. We would use constraint operations such as conjunction and disjunction as oid functions whose semantics is specified by means of constraint simplification. These created constraints are also mapped with $\mathcal{R}$ to the appropriate CST object. For example, if we have two constraints, $2\underline{x} - 4\underline{y} + 5\underline{z} \leq 30$ and $12\underline{u} + 8\underline{y} - 9\underline{z} \leq -80$, their conjunction

$$2\underline{x} - 4\underline{y} + 5\underline{z} \leq 30\ \wedge\ 12\underline{u} + 8\underline{y} - 9\underline{z} \leq -80$$

is considered as an oid function generating a new oid (which with no simplification whatever in this case), that is $2\underline{x} - 4\underline{y} + 5\underline{z} \leq 30\ \wedge\ 12\underline{u} + 8\underline{y} - 9\underline{z} \leq -80$ and which is mapped to the appropriate collection of points in 4-dimensional space:

$$\{\ a_1, a_2, a_3, a_4\ :\ 2a_1 - 4a_2 + 5a_3 \leq 30\ \wedge\ 12a_4 + 8a_2 - 9a_3 \leq -80\ \}$$

In queries new CST objects will be created by means of using *disjunctive existential* constraints in the SELECT clause.

3.3 CST Classes

CST objects are instances of CST classes, whose attributes and methods define the information and operations attached to the infinite collections of points they represent. CST classes are distinguished by their dimension. Thus, for each $k \geq 1$, we have the class $CST(k)$ which represents constraints defining collections of points in $\Re^k$. Every class of CST objects that represent collections of points in $\Re^k$ is a subclass $CST(k)$. For example, the linear constraint, $2\underline{x} - 4\underline{y} + 6\underline{z} \leq 6$, is an instance of that class, and represents the infinite collection of points, $\{ a_1, a_2, a_3 \ : \ 2a_1 - 4a_2 + 6a_3 \leq 6 \}$, in three-dimensional space. Note that it is also an instance of $CST(j)$, for every $j \geq k$.

The operations of CST superclasses define the constraint operations such as intersection or union as methods. These methods are polymorphic in that they have multiple signatures. As an example for the usage of constraint classes, consider the class *Drawer*:

$$CLASS \quad Drawer \quad [\quad color \ : \ Color; \quad extent \ : \ CST(2); \ ]$$

whose `extent` attribute is over CST classes, and describe te drawer's location and volume, respectively. An instance of this class could be:

$$d \ : \ [\ color \ \rightarrow \ 'red'; \quad extent \ \rightarrow \ 2\underline{x} - 4\underline{y} \leq 45 \ ]$$

Observe that we regard an attribute over a CST class such as `location` as a single-valued and not as a multi-valued attribute. This corresponds to our view of a CST-object as a first-class object in the data model.

Given the semantics of constraints as described in section 3.1, an instance of class $CST(k+1)$ is also an instance of the class $CST(k)$ for every $k \geq 1$. Hence, we define $CST(k+1)$ to be a subclass of $CST(k)$. The different canonical forms of constraints can be used to further specialize constraint classes in order to limit operations allowed on constraints within the $\mathcal{L}yri\mathcal{C}$ language to tractable complexity classes. Such limitations can be enforced by means of a type checking mechanism of $\mathcal{L}yri\mathcal{C}$ that is similar to that of XSQL [KKS92].

One can place additional attributes and methods on subclasses of the above mentioned constraint classes in order to capture additional information. For example, in the maneuver decision aid we would have a CST class corresponding to regions in which we would add a *rating* attribute giving the rating of the maneuver region:

$$CLASS \ Region \ SUBCLASS \ CST(4) \quad [\quad rating \ : \ integer \quad]$$

Note that the names of variables do not participate in any way in defining the class to which a particular constraint belongs. They will be used only in database schema definitions as we next discuss.

3.4 Schemas of Constraint Databases

We discussed above the usage of CST classes to represent CST objects. We next turn to their usage in database schemas to represent spatio-temporal information on other objects. This is done by means of classes in which some attributes are over a CST class such as the `location` attribute of the class *Drawer* above. While such classes capture the right semantics of such spatio-temporal information, the user would like stronger means to describe CST information. First, we would like the variables in constraints to act as logical variables in the sense that their names are significant and using the same variable name in different constraints implies the need to substitute the same value. Moreover, attributes of different classes are often jointly constrained such as the `drawer_center` attribute of a desk and the `extent` attribute of the desk's drawer.

Accordingly, we allow the user to list the variables that appear in the constraints assigned to different attributes over CST classes. These variables will be used to facilitate constraint manipulation. Thus, for a constraint class $CST(k)$ we would place in parenthesis the names of the variables as, $CST(x_1, \ldots, x_k)$. Figure 1 shows an example of a class in which the CST attributes include a specification of the variables to be used in them in addition to the dimension of the CST objects (for the moment ignore the variable attached to reference attributes and those attached to class names). Note that an attribute over a CST class may be set-valued, e.g. `drawer_center` attribute in the class *File_Cabinets* Fig. 1.

To enable the user to specify a database in a modular way, we have to allow for constraints in different parts of the database to use the same names for variables. Thus, variables in separate classes may have the same name without implying the projection of the two CST objects which are assigned to these attributes over the same coordinate have the same value. This can be regarded as a simple extension of the usual approach in object-oriented models in which distinct attributes in different classes may have the same name. For example, the same variables can be used in defining schema for the `extent` of a $Office\_Object$ and for the `extent` of a drawer of an object. However, each CST attribute is used to represent a distinct collection of points.

Occurrence of the same variable name in the definition of two constraint attributes of the same class is significant in the following. It implies that for a point in one of them there is a corresponding point in the other with the same value in the coordinate corresponding to the same variable if both attributes are conjoined in a $\mathcal{L}yri\mathcal{C}$ query (see 4 below). Effectively, it would introduce an implicit equality constraint if the CST attributes of the same object appear as arguments of a constraint operations in a query. For example, `translation` and `extent` attributes of $Office\_Objects$ use the same w, z variables so that they translate the `extent` from one coordinate system to another. If we want to translate the `extent` of an object expressed in the local coordinates into the room's coordinates by means of conjuncting it with the `translation` attribute, we would like to enforce the equality between the corresponding arguments of these predicates by using the same names w, z in their schema. Although we can always express equalities in the query itself, we would often want to capture some of the equalities at the level of the schema and thereby automatically get thme in the query.

We would often require inter-object constraints, in particular to relate various attributes of an object and its parts. These mutually constrained objects may belong to different classes. In our running example, the `extent` attribute of the *drawer* in the desk's coordinates will be restricted by its `translation` attribute and by the `drawer_center` attribute of the desk, expressing the flexibility of the movement of the drawer in its tracks.

These shared variables in a schema only imply a possibility that they will have the same value, but this possibility will be used only if the same attributes will be retrieved in the same query, and will appear within the same constraint formula used inside the SELECT or WHERE clauses. Thus, their usage is analogous to the use of the same attribute name to facilitate a natural join between different relations in a relational database.

To preserve modularity of classes, both in terms of freedom to list variables in distinct classes independently, and the ability to delimit the objects referencing instances of the class to constrain its attributes, we would introduce an interface mechanism for classes in which CST attributes are used. For each such class, we would have an interface listing the variables used in its attributes which may be possibly constrained in attributes of objects referencing that class (i.e. that have attributes ranging over the class). For a class C, its interface is specified by attaching a list of the variables that can be externally constrained to its name, i.e. $C(x_1, \ldots, x_n)$. A class C' which has an attribute A over class C may rename its interface as in $A : C'(y_1, \ldots, y_n)$. This allows its CST attribute to use variables independent of those of class C. As an example, see the class $File\_Cabinet$ in Figure 1 where the `drawer` attribute renames the interface of the $Drawer$ class, and then constrains it in the attribute `drawer_center`

Consider a room with a desk depicted in Figure 2.
The following is the constraint description of the object *my-desk* in the room:

my_desk.inv_number	22-354
my_desk.location	$((x, y) \mid (x = 6 \wedge y = 4))$
my_desk.catalog_object[co]	
co.name	'standard_desk'
co.color	'red'
co.extent	$((w, z) \mid (-4 \leq w \leq 4 \wedge -2 \leq z \leq 2))$
co.translation	$((w, z, x, y, u, v) \mid (u = x + w \wedge v = y + z))$
co.drawer_center	$((p, q) \mid (p = -2 \wedge -2 \leq q \leq 0))$
co.drawer[D]	
d.extent	$((w, z) \mid (-1 \leq w \leq 1 \wedge -1 \leq z \leq 1))$
d.translation	$((w, z, x, y, u, v) \mid (u = x + w \wedge v = y + z))$

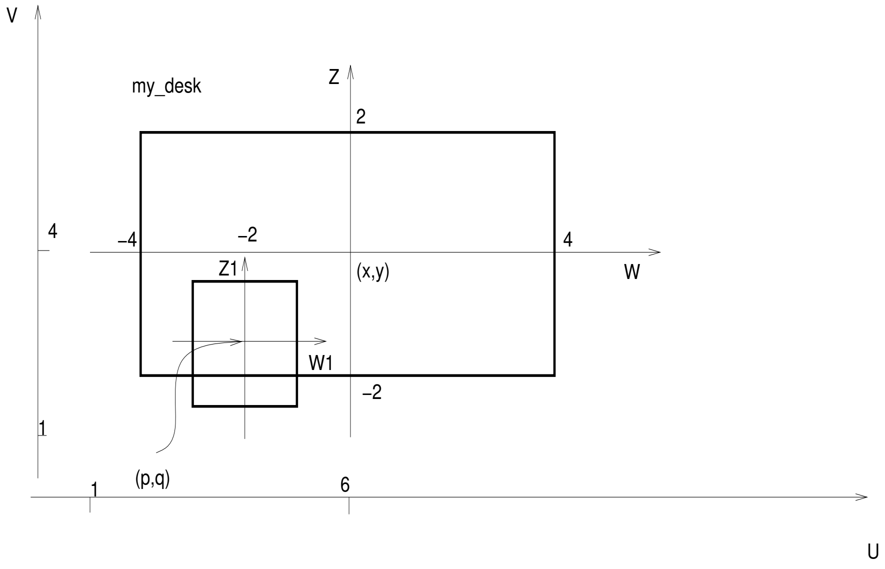

Figure 2: An instance of an object in the room

4 The $\mathcal{L}yri\mathcal{C}$ Query Language

The $\mathcal{L}yri\mathcal{C}$ query language is intended to query constraint object bases in our data model. Before formally defining the $\mathcal{L}yri\mathcal{C}$ syntax and semantics, we first explain them intuitively through a number of examples.

4.1 $\mathcal{L}yri\mathcal{C}$ by Examples

Path expressions in $\mathcal{L}yri\mathcal{C}$ have the same syntax and semantics as those of XSQL. For instance, the path expression,

$$\texttt{standard\_desk.drawer.extent} \tag{3}$$

describes a path that starts in the object of class *Office_Objects* denoted by *standard_desk*, continues to the `drawer` of that desk and ends in `extent` of that drawer. A similar path expression can be utilized in the following query to retrieve all `extent` attributes of drawers in desks, which form constraints:

```
SELECT Y
FROM Desk X
WHERE X.drawer.extent[Y]
```

This query treats CST objects purely as logical oids. For a *standard_desk* object that appears in the example database of Section 3, this query will return the expression,

$$((w,z)|(-1 \leq w \leq 1 \ \wedge \ -1 \leq z \leq 1))$$

which is a logical oid of the extent of the drawer of the *standard_desk*.

The following $\mathcal{L}yri\mathcal{C}$ query returns, for each catalog object, its extent in the global (room) coordinates, assuming its center is located at the point $(6,4)$.

`SELECT CO,` $((u,v) \mid (E(w,z) \wedge D(w,z,x,y,u,v) \wedge x = 6 \wedge y = 4))$
`FROM Office_Object CO`
`WHERE CO.extent[E] and CO.translation[D]`

First note that the FROM and WHERE clauses here are not different from those in XSQL. For each instantiation of oids into variable selectors CO, E, and D, consistent with the FROM clause and the path expressions in the WHERE clause, the query produces a tuple of two oids: *co* for the catalog object and another one, for the extent in the room coordinates, expressed by, $((u, v) \mid (e(w, z) \wedge D(w, z, x, y, u, v) \wedge x = 6 \wedge y = 4))$. We can view this expression as defining a collection of all points (u, v) such that there exist w, z, x, y such that (w, z) satisfy e (meaning it is in the extent of the desk) and such that w, z, x, y, u, v satisfy the equation in d, and such that $x = 6$ and $y = 4$. Note that the syntax and semantics here are very close to those of the relational calculus. As in the relational calculus, the CST expressions in $\mathcal{L}yri\mathcal{C}$ queries are invariant to variable names used. It might be convenient though to use the variables names appearing in a schema. Recall now that in the database schema in Figure 1, the same variables (w, z) are used in the description of `extent` and `translation` of the same object. This lets us rewrite the above query in a shorter form using the implicit equation introduced by variable names:

SELECT CO, $((u, v) \mid (E \wedge D \wedge x = 6 \wedge y = 4))$
FROM Office_Object CO
WHERE CO.extent[E] and CO.translation[D]

To see why the above mentioned expression correspond to what we need to find, observe that $D(w, z, x, y, u, v) \wedge x = 6 \wedge y = 4$, gives equations describing the connection between local (object) coordinates (w, z) of a point, and its global coordinates (u, v) assuming the center of the object is at $(6, 4)$. For instance, for the catalog object *co* corresponding to `my_desk` from the example database (i.e. `my_desk.catalog_object[co]`, $D(w, z, x, y, u, v) \wedge x = 6 \wedge y = 4$, is $(u = x + w \wedge v = y + z) \wedge x = 6 \wedge y = 4$, or, equivalently, $u = 6 + w \wedge v = 4 + z$. Then $(e(w, z) \wedge D(w, z, x, y, u, v) \wedge x = 6 \wedge y = 4))$ would become $(-4 \leq w \leq 4 \wedge -2 \leq z \leq 2) \wedge u = 6 + w \wedge v = 4 + w$. Thus, if we project on (u, v), i.e. consider $((u, v) | (-4 \leq w \leq 4 \wedge -2 \leq z \leq 2) \wedge u = 6 + w \wedge v = 4 + w)$, $\mathcal{L}yri\mathcal{C}$ would simplify it to $((u, v) | 2 \leq w \leq 10 \wedge 2 \leq v \leq 6)$, which is exactly the extent of the standard desk in the global coordinates assuming its center is at $(6, 4)$. To verify this see Figure 2. The symbolic expression $((u, v) | 2 \leq w \leq 10 \wedge 2 \leq v \leq 6)$ (again invariant to variable names) is a logical oid to be returned in the SELECT clause.

The following $\mathcal{L}yri\mathcal{C}$ query answers the following question. Assuming the room is 20×10, for each desk whose center may appear in the left upper quater of the room, find the area that can be occupied by its drawer (in any position) in the room's coordinates:

SELECT O, $((u, v) | (D(w, z, x, y, u, v) \wedge DD(w1, z1, x1, y1, u1, v1) \wedge w = u1 \wedge z = v1$
$\wedge DC(p, q) \wedge DE(w1, z1))$
FROM Object_In_Room O, Desk DSK
WHERE O.location[L] and $((L(x, y) \wedge 0 \leq x \leq 10 \wedge 5 \leq y \leq 10))$ and
O.catalog_object[DSK] and
DSK.translation[D] and
DSK.drawer_center[DC] and
DSK.drawer.translation[DD]and
DSK.drawer.extent[DE]

The expression :

$$((u, v) | (D(w, z, x, y, u, v) \wedge DD(w1, z1, x1, y1, u1, v1) \wedge (w = u1) \wedge (z = v1) \wedge$$
$$L(x, y) \wedge DC(p, q) \wedge DE(w1, z1))$$

gives an area (described in the global coordinates) of all points that can be occupied by the drawer of a desk. Here too we have implicit equalities derived from the schema. Namely, since in the schema the attribute `drawer` of the desk is "invoked" with actual parameters (p, q), while the "formal" parameters are (x, y), we must have an equality stating that the first and second arguments of `DSK.drawer_center` must be equal to the third and fourth arguments of `DSK.drawer.translation` correspondingly. In the query it will translate to the equalities $p = x1 \wedge q = y1$. Thus the CST expression to be evaluated in the SELECT clause is in fact

$$((u, v) | (p = x1 \wedge q = y1 \wedge \ D(w, z, x, y, u, v) \wedge DD(w1, z1, x1, y1, u1, v1) \wedge w = u1 \wedge z = v1 \wedge$$
$$L(x, y) \wedge DC(p, q) \wedge DE(w1, z1))$$

Note also that $D(w, z, x, y, u, v) \wedge DD(w1, z1, x1, y1, u1, v1) \wedge w = u1 \wedge z = v1$ describes equations relating coordinates $(w1, z1)$ of a point described in the drawer's system with the coordinates (u, v) of the same point described in the global coordinate system.

The condition $((L(x, y) \wedge 0 \leq x \leq 10 \wedge 5 \leq y \leq 10))$ in the WHERE clause is true, if the logical formula corresponding to it is *satisfiable*, i.e. there exist a real number substitution into the variables that makes the formula true. In our query, it would mean that there exist a possible location of the DSK (x, y) that satisfies $((x, y)|0 \leq x \leq 10 \wedge 5 \leq y \leq 10)$, meaning that the point in the left upper quater of the room.

The next $\mathcal{L}yri\mathcal{C}$ query gives, for each red desk in the catalog with a drawer in the middle of the table, its extent above the 45 degree line through its center.

```
SELECT DSK, ((w,z)|(DSK.drawer.extent(w,z) ∧ z ≥ w))
FROM Desk DSK
WHERE DSK.color = 'red' and DSK.drawer_center[C] and (C(p,q) ⊨ p = 0)
```

Here we use CST predicate $\models$ in the WHERE clause. Its meaning is the standard one in logic, namely, it is true if for all real numbers p, q, $C(p, q) \Rightarrow p = 0$, i.e. every possible center of the drawer must be in the middle of the desk.

The following query finds all desks in the room whose drawer does not touch the walls of the room, assuming the room is 20×10.

```
SELECT DSK
FROM Object_In_Room O, Desks DSK
WHERE O.catalog_object[DSK]
      DSK.drawer_center[C] and
      DSK.translation[D] and
      DSK.drawer.extent[DRE] and
      DSK.drawer.translation[DRD] and
      (C(p,q) ∧E(w,z) ∧DRD(w1,z1,x1,y1,u1,v1) ∧ D(w,z,x,y,u,v)
             ∧w = u1 ∧z = v1  0 < u < 20 ∧ 0 < v < 10)
```

By combining higher-order variables with constraints, we can define powerful views that classify objects according to their spatial attributes. Assume a class *Region* of office regions which is a subclass of $CST(2)$, we can classify the objects in a design into different classes, according to the element of *Region* in which they are places. This is done by means of the following view:

```
CREATE VIEW X AS SUBCLASS OF Object_In_Room
SELECT X FROM Object_In_Room Y, Region X
WHERE X.extent[U] and (U ⊨ X)
```

Note that X is used in the CREATE clause as the oid of the class, in the FROM clause as the oid of an object, and in the WHERE clause as a constraint.

4.2 Syntax and Semantics of $\mathcal{L}yri\mathcal{C}$

The syntax of $\mathcal{L}yri\mathcal{C}$ query language is a superset of XSQL. Therefore we only briefly describe the additions, which include a number of operators to create new CST objects in the SELECT clause and a number of predicates to be applied to CST objects in the WHERE clause. We will call all variables in the query, that are not variable selectors, *constraint variables*.

Pseudo-Linear formula is an arithmetic expression that may involve constraint variables, constants and path expressions that, when instantiated, define constants of type *real* or *integer*. The formula is required to be *pseudo* linear in the sense that when all non-constraint variables are instantiated, the formula must be representable as a *linear arithmetic* constraint.

We define now CST *formulas*. Specifically, *conjunctive*, *disjunctive*, *existential conjunctive*, and *disjunctive existential* formulas are defined as extensions of the corresponding types of constraints defined in Section 3 as follows. First, we allow *pseudo-linear* formula everywhere that a *linear arithmetic constraint* is allowed. Second, everywhere in the definitions where we allowed *conjunctive*, *disjunctive*, *existential conjunctive*, or *disjunctive existential* constraints, we will also allow an expression of the form, $O(x_1, \ldots, x_n)$, or of the form O where O is a path expression, that, when instantiated, defines an n-dimensional CST object of the corresponding (conjunctive, disjunctive etc.) type, and where $x_1, \ldots, x_n$ are constraint variables. If the variables are not specified, they are simply copied from the schema.

Since a CST object is defined as an interpreted predicate, the truth value of O for an instantiation of real numbers into $x_1, \ldots, x_n$ is defined. Thus, when all non-constraint variables are instantiated, the truth value assignment of a CST *formula* is defined exactly as for the constraints, while taking into account the truth values of the CST predicates (objects).

$\mathcal{L}yri\mathcal{C}$ allows the following additional types of attributes in the SELECT clause:

1. An *disjunctive existential* formula of the form $((x_1, \ldots, x_n) \mid \phi)$
   This formula will define an n-dimensional CST object as an interpreted n-dimensional predicate, and its logical oid as the required canonical form of constraints.

2. An expression of the form, `MAX(`f `SUBJECT TO` $(x_1, \ldots, x_n)|\phi$`))`, or
   `MIN(`f `SUBJECT TO` $(x_1, \ldots, x_n)|\phi$`))`, where $f(x_1, \ldots, x_n)$ is an objective function described as a linear combination of constraint variables, and ϕ is an *existential conjunctive* formula. The meaning of this operator is a linear programming problem of finding maximum or minimum of a linear objective function subject to a system of linear constraints.

3. An expression of the form `MAX_POINT(`f `SUBJECT TO` $(x_1, \ldots, x_n)|\phi$`)` or
   `MIN_POINT(`f `SUBJECT TO` $(x_1, \ldots, x_n)|\phi$`)`, with the same meaning for f and ϕ, to find a point in n-dimensional space at which the maximum, or, respectively, minimum is achieved.

In the WHERE clause, $\mathcal{L}yri\mathcal{C}$ also allows the following constraint predicates:

1. A *satisfiability* predicate in the form of a *disjunctive existential* formula ϕ. This predicate gets the value `true` in the WHERE clause if ϕ is satisfiable, i.e. there exist instantiations of real numbers into its free constraint variables that makes the formula `true`.

2. An expression of the form:
   $$((x_1, \ldots, x_n)|\phi) \models ((y_1, \ldots, y_m)|\phi')$$
   where ϕ and ϕ' are *disjunctive* formulas. This $\models$ predicate gets the value `true` in the WHERE clause if for every real numbers instantiation into $x_1, \ldots, x_n, y_1, \ldots, y_m$, the truth value of $((x_1, \ldots, x_n)|\phi)$ implies the truth value of $((y_1, \ldots, y_m)|\phi')$

The semantics of a LyriC query is defined as an extension of that of XSQL. Hence, we just describe those aspects of $\mathcal{L}yri\mathcal{C}$ that pretain specifically to constraints. To evaluate CST predicates *satisfiable* or *implies*, we first add implicit equality constraint derived from the schema. Then, the truth value of the CST operators is evaluated as explained earlier. The Boolean operators (*and, or,* and *not*) are evaluated in the usual way. To create an oid of a new CST object, we first add implicit constraint derived by the schema. Then, we evaluate the oid as explained earlier. One can generate new objects some of whose attributes are constraints by using oid functions as in XSQL, as constraints are regarded as another kind of oid.

5 Complexity of $\mathcal{L}yri\mathcal{C}$ Query Evaluation

To show the tractability of evaluating $\mathcal{L}yri\mathcal{C}$ queries we show how to translate them into SQL with linear constraints [BJM93]. Note that the definition of a database in $\mathcal{L}yri\mathcal{C}$ as a general structure (see section 3.2) means that it is essentially a collection of flat relations [KLW90]. These represent the extent of classes and the mapping used to represent attributes. We assume in the translation that methods are not employed in queries as they provide unlimited computational power. We next join the class relations, the single-valued attribute relations, and the the multi-valued attribute relations (after unnesting them) together, obtaining a flat relation for each class in the database.

We next translate a $\mathcal{L}yri\mathcal{C}$ query into a SQL query with constraints [KKR93]. We first flatten all path expressions into a single level by the addition of class names and variables in the FROM clause. Thus, the language is equivalent to SQL with linear constraints and hence has a PTIME data complexity.

A more sophisticated implementation that we plan to develop would be through a constraint algebra in which higher-order operators manipulate collections of objects (e.g. sets, lists) some of whose elements may be constraints. Thus, the algebra is an FP-like language [Bac78, BK93] in which functional forms capture common data collections processing abstractions such as filtering elements, and applying a function to all elements of a collection, and primitive functions manipulate objects of different types such as intersecting constraints. Since constraint database optimization considerably differs from that of regular databases, the algebra will have to accommodate some new optimization frameworks, such as the one in [BJM93]. A full description of the algebra, the translation of $\mathcal{L}yri\mathcal{C}$ into it, and an algebraic optimizer is an issue of future research.

6 Related Work

No technology for declarative and efficient querying in the target CST applications exists today. Existing tools for constraint manipulation are built for specialized applications and are not well integrated with available DBMS. This causes an impedance mismatch in terms of query manipulation, and prevent efficient implementation of query evaluation over constraint databases. Existing DBMS do not deal with and manipulate constraints as stored data [2]. Constraint Logic Programming [JaL87, CHIP, Prolog3] on the other hand was not designed to deal with large amounts of stored data, and support spatio-temporal features. Extensions of DBMS with spatio-temporal operators [OrM88, Gut89, HaC91] are typically limited to low (two- or, at most three-) dimensional space, have restrictions on using these operators in query languages, and lack global economical filtering and deep optimization.

The work [KKR93] proposed a framework for integrating abstract constraints into database query languages by providing a number of design principles and studied important properties of specific instances of the framework. However, they did not consider languages supporting complex objects and did not focus on optimization. The work [HHLB89] considered more than just linear constraints (but only equalities). However, reasoning with the constraints was limited to local propagation steps, and hence is not practical for linear programming problems. A restricted form of linear constraints, called *linear repeating points*, was used to model infinite sequences of time points [KSW90, BNW91, NS92]. More recent works on deductive databases [MFPR, SrR92, KS93, LS92] have attempted to extend optimization methods for Datalog to cope with constraints. However, that research concentrated on optimizing recursion and repositioning of constraints, and assume as given, the implementation and optimization of the basic relational database operations involving constraints. The work [BJM93] introduced *Linear Constraint Databases*, concentrating on their optimization, and proposed a new generic optimization framework.

Srivastava, Ramakrishnan and Revesz [SRR94] have proposed an integration of constraints into an object-oriented data model. They suggest that attributes whose values are only partially known to be specified using constraints. In contrast, in our model, constraints are used as a complete specification of a CST object (even without universal quantification over variables). This different philosophy, implies that their concerns about the different possible semantics of incomplete information, represented as constraints, are irrelevant to us, and their model serves different applications than ours. In their model constraints are integrated into the model in a more limited way than in $\mathcal{L}yri\mathcal{C}$, where they are first-class citizens in a very general object-oriented data model. Moreover. we avoid the limitations of regarding the database as a global constraint as discussed in section 3.4 above. We are concerned just with linear constraints because they offer a tractable and practically expressive class of constraints. Basing $\mathcal{L}yri\mathcal{C}$ on SQL seems to us as a more practical and complete way to query databases than declarative query languages such as COQL. Updating CST attributes is completely general as it should be in the required applications, and is not limited to monotonicly increasing refinement of constraint attributes as in their model (e.g. there is no reason that moving a desk would be limited in any way).

[2] Note, integrity constraints used in conventional databases are not data, but rather something the data must satisfy.

Acknowledgement: We are indebted to Paul Exarkhopoulo, and Dimitra Vista for helpful advice and comments, and Ami Motro for suggesting the name $\mathcal{L}yri\mathcal{C}$.

References

[AB91] Abiteboul, S., A. Bonner, "Objects and Views," *Proc. ACM SIGMOD Conf. on Management of Data*, 1991.

[AK89] Abiteboul, S., P. C. Kanellakis, "Object Identity as a Query Language Primitive," *Proc. ACM SIGMOD Conf. on Management of Data*, 1989, pp. 143–153.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BEER89] Beeri, C., "Formal Models for Object-Oriented Databases," *Proc. First Int. Conf. on Deductive and Object-Oriented Databases,* Kyoto, Japan, Dec. 1989, pp. 370–395. *IEEE Trans. on Knowledge and Data Engineering,* Dec. 1989.

[BJM93] A. Brodsky, J. Jaffar, M.J. Maher, Toward Practical Constraint Databases. *Proc. 19th International Conference on Very Large Data Bases,* Dublin, 1993. pp. 322-331, Atlantic City, May 1990.

[BK93] C. Beeri, Y. Kornatzky. Algebraic optimization of object-oriented query languages. *Theoretical Computer Science*, 116(1), 1993.

[BNW91] M. Baudinet, M. Niezette, P. Wolper, On the representation of infinite temporal data and queries, *Proc. ACM Symp. on Principles of Database Systems*, 1991.

[BVCS93] M. Benjamin, T. Viana, K. Corbett, A. Silva, Satisfying Multiple Rated-Constraints in a Knowledge Based Decision Aid, *Proc. IEEE Conf. on Artificial Intelligence Applications,* Orlando, 1993.

[CHIP] M. Dincbas, P. Van Hentenryck, H. Simnis, A. Aggoun, T. Graf, F. Berthier, The Constraint Logic Programming Language CHIP, *Proc. Fifth Generation Computer Systems,* Tokyo Japan, 1988.

[CKW89] W. Chen, M. Kifer, and D.S. Warren, HiLog: A First Order Semantics for Higher-Order Logic Programming Constructs, *In 2-nd Intl. Workshop on Database Programming Languages*, Morgan-Kaufmann, june 1989.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[Gut89] R.H. Guting, GRAL: An extensible relational database system for geometric applications, *Proc. 19th Symp. on Very Large Databases,* 1989.

[HaC91] L.M. Haas, W.F. Cody, Exploiting extensible DBMS in integrated geographic information systems, *Advances in Spatial Databases*, Proc. 2nd Symp. on Spatial Databases, Lecture Notes in Computer Science 525, Springer Verlag, Berlin, 1991.

[HHLB89] M.R. Hansen, B.S. Hansen, P. Lucas, P. van Emde Boas, Integrating Relational Databases and Constraint Languages, *Computer Languages* 14, 2, 63–82, 1989.

[HMS92] N. Heintze, S. Michaylov & P.J. Stuckey, CLP($\mathcal{R}$) and Some Electrical Engineering Problems, *Journal of Automated Reasoning* 9, October 1992, 231-260.

[JaL87] J. Jaffar, J-L. Lassez, Constraint Logic Programming, *Proc. Conf. on Principles of Programming Languages*, 1987, 111–119.

[JMSY92] J. Jaffar, M.J. Maher, P.J. Stuckey & R.H.C. Yap, Output in CLP($\mathcal{R}$), *Proc. Int. Conf. on Fifth Generation Computer Systems 1992,* Tokyo, Japan, Vol. 2, 1992, 987–995.

[KKR93] P. Kanellakis, G. Kuper, P. Revesz, Constraint Query Languages, *Journal of Computer and System Sciences*, to appear. (A preliminary version appeared in *Proc. 9th PODS*, 299–313, 1990.)

[KKS92] M. Kifer, W. Kim, Y. Sagiv. Querying object-oriented databases. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 393–402, 1992.

[KLW90] Kifer, M., G. Lausen, J. Wu, "Logical Foundations of Object-Oriented and Frame-Based Languages," Technical Report #90/14, Department of Computer Science, SUNY at Stony Brook, August 1990. to appear in J. of ACM.

[KS93] D. Kemp, P. Stuckey, Bottom Up Constraint Logic Programming Without Constraint Solving, Technical Report, Dept. of Computer Science, University of Melbourne, 1992.

[KSW90] F. Kabanza, J.-M. Stevenne, P. Wolper, Handling infinite temporal data, *Proc. ACM Symp. on Principles of Database Systems*, 1990.

[KW89] Kifer, M., J. Wu, "A Logic for Object-Oriented Logic Programming (Maier's O-Logic Revisited)," *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems,* 1989, pp. 379–393.

[KV84] G.M. Kuper and M.Y. Vardi. A new approach to database logic. In *Proc. Third ACM Symp. on Principles of Database Systems*, pages 86–96, 1984.

[LS92] A. Levy, Y. Sagiv, Constraints and Redundancy in Datalog, *Proc. 11-th PODS*, 67-80, 1992.

[MFPR] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, R. Ramakrishnan, Magic Conditions, *Proc. 9th PODS*, 314–330, 1990.

[NS92] M. Niezette and J.-M. Stevenne, An efficient symbolic representation of periodic time, *Proc. of First International Conference on Information and Knowledge management*, 1992.

[OrM88] J.A. Orenstein, F.A. Manola, PROBE spatial data modeling and query processing in an image database application, *IEEE Trans. on Software Engineering 14*, 5, pp. 611–629, 1988.

[Prolog3] A. Colmerauer, An Introduction to Prolog 3, *CACM,* 33:7:69-90,1990.

[Sri92] D. Srivastava, Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints, *Annals of Mathematics and Artificial Intelligence*, to appear.

[SrR92] D. Srivastava, R. Ramakrishnan, Pushing Constraint Selections, *Proc. 11th PODS*, 301–315, 1992.

[SRR94] D. Srivastava, R. Ramakrishnan, P. Revesz, "Constraint Objects", *Proc. 2nd Workshop on the Principles and Practice of Constraint Programming*, Orcas Island, WA, May 1994.