Pictorial Deduction in Spatial Information Systems

Bernd Meyer
FernUniversität Hagen, Praktische Informatik IV, 58084 Hagen, Germany
bernd.meyer@fernuni-hagen.de

Abstract

Though visual access to spatial database systems has attracted much attention in recent years, there have only few deductive visual languages for spatial information systems been proposed. One reason for this may be that most of the spatial visual languages stick to the classical metaphor of map manipulation. We argue that more powerful metaphors are needed to facilitate the development of pictorial languages for spatial deduction that are readily comprehensible and formally manageable at the same time. This paper introduces a fully deductive pictorial language for spatial information based on sketching.

1 Introduction and Motivation

In the last decade spatial information systems in general and geographical information systems in particular have become an ever growing field of research. In recent years visual interfaces to spatial information systems have attracted an increasing amount of interest. Several proposals have been made and working systems have been implemented. Some of the most important approaches are described in [1–6]. Visual information access seems to be even more natural for spatial information than for general, abstract information, since spatial information can obviously be mapped to pictorial representations: spatial properties in the real world can directly be depicted as spatial properties in the visualization space, or, to put it shorter, space is depicted as space.

On the other hand, deductive mechanisms, which are a well investigated technology in the area of standard information systems, have rarely been discussed for visual spatial query languages, though there are several approaches to spatial reasoning formalisms (A brief overview can be found in [7]). To our knowledge the only real visual deductive spatial query language is presented in [8]. This surprisingly small amount of attention may be partially due to the fact that the deductive, i.e. rule-based, paradigm is not well in-line with the system metaphor that is mostly used: cartographic manipulation. Another reason may be that it is very difficult to represent universal, disjunctive, or negative information in diagrams. But these types of information are exactly what is required in a deductive system.

Our claim is that new system metaphors are needed to make pictorial deduction in spatial information systems feasible. This paper will present a deductive visual query language (Sketch) for spatial information systems whose central metaphor is sketching. Input sketches are interpreted by the query system and are translated into expressions of an object-oriented logic calculus. A first, non-deductive version of this language has been published in [9]. As Sketch is based on a logic approach, it lends itself to be enhanced by deductive capabilities. The objective of this paper is to present the deductive extensions of the visual language and the data model.

1.1 A First Glance at Sketched Queries

The sketch in Figure 1 shows a simple example of a visual query. Assume there is high water in a certain lake. The query asks for the names of all people who might be hindered to get to their place of work, because the way from their home to this place may be flooded. This is the case if this way has to cross a river that runs to or from the lake.

![Figure 1: A First Example](image)

The interpretation of the query in terms of the object calculus is:

\[
P1: \text{person} \land T1: \text{town} \land Rd1, \text{Rd2}: \text{road} \land F1: \text{factory} \\
\land R1: \text{river} \land L1: \text{lake}(\text{level}="\text{high})" \land B1: \text{bridge} \\
\land W: \text{works-at}(\text{employee}=P1, \text{place}=F1) \\
\land L: \text{lives-in}(\text{resident}=P1, \text{location}=T1) \\
\land R1 \text{ intersects } L1 \land R1 \text{ intersects } B1 \land \text{Rd1 touches } F1 \\
\land \text{Rd1 touches } B1 \land \text{A1 touches } \text{Rd2} \land \text{Rd2 touches } T1.
\]

Note that the basic structure of the formula is to some extent Datalog-like, but offers additional support for objects, datatypes, and spatial operators. The formula given above can be fulfilled if there are certain objects (line 1-2) that satisfy the spatial properties depicted in the sketch (line 5-6) and participate in the abstract relationships given by the graph (line 3-4). A more detailed explanation of the calculus will be given later. It is important to realize that every sketch has a single well-defined interpretation which can formally be derived from the picture. Thus, every sketch is an exact formal query just like, e.g., an SQL expression or a Datalog formula.
2.1 Basic Concepts

Every object has a unique identity and a number of named attributes, which can either be data-valued (slots) or object-valued (ports). Slots and ports are typed. E.g.

class road: slots: name -> string, route -> line, ports: from -> city, to -> city.

defines a class road whose objects have two data-valued slots (one of which is a spatial attribute) and two object-valued ports. Objects without object-valued attributes are called simple objects, non-simple objects are called links. An object is completely defined by its class, its identity, the values in its slots, and the references in its ports.

Links are the only way to model relations between objects, as there is no separate concept for relationships. Note that links, like simple objects, possess an identity.

The identity of an object is unique only within a single class. Thus, several classes can be used to model different views of the same objects. In combination with rules this will later be used to provide ad-hoc generalizations, specializations, and heterogeneous collections of objects.

Support for spatial information is given by spatial predicates and spatial functions. Predicates accept objects as arguments and are evaluated with respect to specific geometric slots (e.g., Road.route intersects Road2.route). Unambiguous slotnames may be dropped. Spatial functions accept objects as arguments and can either be data-valued (as in \(\text{min\_dist}(\text{Road1}, \text{Road2})<100)\) or object-valued. In the latter case we call them object transformers.

As spatial operations like intersection may have collections of objects as a result (e.g., a polygon and a line may intersect in an arbitrary number of points), object transformers are multiply evaluable in the same sense as predicates in logic programming can be multiply fulfilled. Hence, transformers can yield several results for the same argument combination, but only a single result at a time.

For the remainder of the paper we will assume that the set of spatial predicates is \(F=\{\text{closest, intersects, neighbouring, inside, touches}\}\) and that the set of spatial functions is \(F=\{\text{min\_dist, max\_dist, intersection, common\_border, center}\}\). The semantics of most of the operators should be obvious. Note that predicates can have corresponding functions: \(\text{common\_border}(X, Y)\) calculates the common line segments if \(X \text{ neighbouring} Y\) is true. \(\text{intersection}(X, Y)\) calculates the intersection objects if \(X \text{ intersects} Y\) is true. We will use the above sets of operators for our discussion, because they cover the most fundamental spatial properties, but both, the data model and the implementation, can easily be adapted to support different sets of operators.

2.2 Queries

Queries are formed by conjunctions of object formulas. An atomic object formula is either a predicate application, a restriction, or a declaration of an object variable. A
declaration like $X, Y: \text{person}$, e.g., declares two variables $X$ and $Y$ for objects of class $\text{person}$. Declarations are used to restrict object variables to range over a finite set of possible bindings. Restrictions (e.g., $X(\text{name}="\text{Bill}", \text{age}=\text{33})$) state certain constraints for the slots or ports of an object variable and thereby restrict the set of possible bindings of the object variable.

To use a formula as a query, the answer attributes have to be defined. Only data-valued variables may be used as answer values. The following example of a simple query calculates the names of border towns:

$$\text{query} \ (\text{name}=\text{N}) \leftarrow C_1, \ C_2: \text{country} \land T: \text{town} \ (\text{name}=\text{N}) \land C_1 \text{neighbouring} \ C_2 \land B: \text{common-border}(C_1, C_2) \land T.\text{location} \text{closest} \ B.$$  

Note that no explicit knowledge of borders is required. The answer is computed from geometric information only. Like with Datalog, the answers to a query are the sets of bindings for the variables in question for which the body of the query is true. The basic form of query evaluation is similar to the bottom-up evaluation of Datalog queries.

3 An Introduction to Sketch: A Visual Query Language Based on Sketching

This section introduces the basic concepts of our visual query language. Only those concepts relevant to the discussion of deductive features will be mentioned.

3.1 The Blackboard Metaphor

The foremost design goal for the visual query language was a language structure that mimics everyday communication processes used in discussing spatial situations, e.g., when someone explains a way through a town or when a teacher talks about geometry. Typically, in these situations example sketches are used which roughly approximate the relevant spatial information that has to be conveyed. Non-spatial information like names is given apart from the sketch in linguistic form.

We have adopted the same schema for the visual query language. Sketch strictly divides queries in non-spatial (propositional) and spatial conditions. While propositional constraints are given as object-graphs, spatial constraints are given in the form of sketches. It is important to note that a sketch is not generated by the query system according to spatial constraints given by the user. Rather, it is drawn by the user and interpreted by the system. The spatial constraints are at no time explicitly stated by the user, they are derived as the result of the interpretation of a sketch by the query system. Hence, a sketch is not a partially abstracted map but a visual logic formula whose meaning is independent of any concrete database contents.

3.2 Basic Concepts of the Visual Language

Query Graphs

Propositional information in expressed by query graphs. A query graph consists of object nodes (circles depicting simple objects) and link nodes (diamonds depicting links). Each node is labelled by the name of an object variable which is by default its classname with a number appended to it. The labels must be unique within a single graph. Query graphs need not be connected. Edges can only emanate from link nodes. The meaning of an edge is that the object (or link) node at which the edge starts is the reference in some port of the link object at which the edge ends. The name of this port is used as a label for the edge. Note that nodes depict instances and not classes. Thus, several nodes of the same class may appear in a query graph. The only type of node that does not depict objects from the database is an is-a node. Such a node is used to represent that two different objects must have the same identity. (Remember that the identity of an object must only be unique within a single class).

Restrictions on data-valued attributes can be made in QBE-style attribute boxes, where values or variables can be entered that have to match the attribute value.

The semantics of a query graph is a DeLOM formula that is the conjunction of declarations for every object shown in the graph and restrictions of the ports to those references given by the edges in the graph. Additionally, the formula contains all slot restrictions that are given in the attribute boxes. As an example, the semantics of the graph in Figure 2 is:

$$F1: \text{factory}(\text{name}=\text{"ABC"}) \land P1: \text{person} \land E1: \text{employed} \land E1(\text{employee}=P1, \text{place}=F1).$$

Figure 2: A Simple Query Graph

Interpretation of Sketches

Sketches consist of spatial objects and implicit topological relations between them. The objects which are contained in a sketch are either drawn by the user or they are subobjects defined by a topological relation between some explicit objects. Every explicit object in a sketch must have a unique label and they all represent necessarily distinct objects. Subobjects are not entered explicitly, but are derived from other objects and their topological relationships. An intersection of two polygonal shaped objects, e.g., implicitly defines at least one polygonal shaped subobject: the common area of both objects. These subobjects can in turn participate in topological relationships with other objects and thereby define further subobjects. Like for the nodes in a query graphs, attribute boxes may be used for objects in a sketch.
The semantics of a sketch is a DeLOM formula that is the conjunction of declarations for all objects, calculated declarations for all implicit subobjects, and applications of spatial predicates so that all the topological relationships between visual objects and/or subobjects are defined. A calculated declaration is a declaration in which the range of possible variable bindings is not given by a classname but by an expression, as in $\text{Z: intersection}(X, Y)$. Roughly speaking, the interpretation algorithm works according to the following rules:

1. Topological relationships are tested for all possible pairs of objects and subobjects in the sketch.
2. All relationships in the set $P$ of spatial predicates are tested.
3. If a predicate $p(x, y)$ holds and there is a function $f \in \mathcal{F}$ corresponding to $p$ then the subobjects will be calculated by $f(x, y)$ and will be interpreted like normal objects.
4. Topological relationships are always tested for an entire object. E.g., it will never be derived that a part of an object is inside of another object.
5. No negative relationships (e.g., not a inside b) will be derived.
6. No relationships between an object and its own subobjects will be derived.
7. Subsumed Subformulas are redundant and can be removed (E.g., $(a \text{ inside } c)$ can be removed from $(a \text{ inside } b \land b \text{ inside } c \land a \text{ inside } c)$).

As an example, the interpretation of the sketch in Figure 3 according to the above rules is:

$$C1, C2: \text{country} \land B: \text{common-border}(C1, C2) \land R1: \text{river} \land R1 \text{ intersects } B \land C1(\text{name} = \text{"Germany"}).$$

To overcome the limitations of expressive power imposed by (4), sketched objects can be transformed by explicit application of spatial transformers like merge, center, vertices or edges. If a transformer is applied, the corresponding subobjects become available in the sketch and are interpreted like normal objects (see Section 5).

A rigorous formal treatment of the semantics of a sketch is given in [10], where the translation of a sketch to a formula is defined by means of Villon, an improved version of picture logic [11].

**Connectives and Negation**

Associations between an object in a query graph and an object in a sketch can be established by using the same label for the node and the visual object. A query may consist of several graphs and several sketches depicted in separate windows. The semantics of a complex query is the conjunction of all formulas derived from all the graphs and sketches. Separate sketches are sometimes indispensable, because by rule (1) and (2) always all relationships are derived. Hence, splitting sketches and thus separating objects is the only way to introduce “don’t-care” relationships between certain pairs of objects.

To handle disjunctions we use a concept of layers: a set of sketch windows and graph windows can be partitioned into several layers. The formulas derived from all the windows in a single layer are combined in a single conjunction and all the layers are combined disjunctively. Negation can be applied either to a single window or to an entire layer. Negation is indicated by crossing out windows (see Figure 4).

Finally we are prepared to turn to pictorial deduction.

### 4 Propositional Deduction

Given a predicative structure as the formal basis of pictorial queries, we can easily extend the language by rule-based views. Such a view defines how a new class can be inferred from other classes. To specify a derived class, we simply name the new class, define its schema, and construct a query that computes the attributes for each member of the new class. In contrast to normal queries, both, data-valued attributes and object-valued attributes are permitted as result attributes. Hence, a class derived by a view can be a simple object class or a link class. The schema definition for a derived class is defined in a way similar to the selection of answer attributes for a query. The user interface provides an attribute editor in the style of the Macintosh Font/DA-Mover-Interface. This editor lets the user select the attributes of the new class among all the ports, slots, and identities of objects used in the query part of the view specification. Each attribute selected defines the contents of one of the slots or ports of the derived class. The name for this slot (port) must be provided by the user whereas its type is inferred from the class definitions of the objects used in the view specification.

As for the data model, DeLOM has to be extended by rule definitions to support views. A rule has the form

$$\text{class}(\text{slot}_1 = X_1, \ldots, \text{slot}_n = X_n, \text{port}_1 = Y_1, \ldots, \text{port}_m = Y_m) \leftarrow \varphi.$$  

where class is the name of the derived class, slot$_1...n$ and port$_1...m$ are its slotnames and portnames, respectively. $\varphi$ is an admissible DeLOM formula, $X_i$ are data-valued variables used in $\varphi$, and $Y_j$ are object-valued variables declared in $\varphi$. The syntactical rules for admissible formulas ensure that such a rule is always safe in the sense of safety in Datalog, i.e., all slot values and port references are defined. Every different set of variable bindings for which $\varphi$ is true defines a new object in class, i.e., if $\theta = \{X_1/v_1, \ldots, X_n/v_n, Y_1/w_1, \ldots, Y_m/w_m, \ldots\}$ is a
variable substitution such that $\varphi\theta$ is true then a new object is derived:

$$\text{object}(\text{class}, \text{new-id}, (\text{slot}_1, v_1), ..., (\text{slot}_n, v_n), (\text{port}_1, w_1), ..., (\text{port}_m, w_m))$$

Some extra attention must be paid to object identities. Is a derived object assigned the same identity as some other object in the database (object preserving semantics) or is it given a unique new identity (object generating semantics)? We allow both kinds of rules. Object generating rules have been outlined above. An object preserving rule has the form

$$\text{class}(\text{slot}_1 = X_1, ..., \text{slot}_n = X_n, \text{port}_1 = Y_1, \text{port}_m = X_m) \text{ is } X \leftarrow \varphi.$$ 

where $X$ is some object variable declared in $\varphi$. Such a rule causes every derived object $O$ to adopt the identity of $X\theta$ where $\theta$ is the set of variable bindings that generates $O$. In a pictorial query, the identity of the derived object has to be defined with an editor similar in appearance to the attribute editor. Object preserving rules are typically used for generalizations or specializations. As an example, look at Figure 4 which defines the rule that every person is a commuter who is not living in the same city in which he works.

![Figure 4: Propositional View with Negation](image)

Figure 4: Propositional View with Negation

$$\text{commuter}(\text{name}=N) \leftarrow P \leftarrow E: \text{person}(\text{name}=N) \land C: \text{city} \land F: \text{factory} \land$$

$$E: \text{estate} \land \text{emp}: \text{employed}(\text{employee}=P, \text{place}=F) \land$$

$$L: \text{live-at}(\text{resident}=P, \text{estate}=E) \land$$

$$\text{not}(\text{F}.\text{location} \text{inside} C \land E \text{inside} C).$$

An object defined by a view can be used in a query like any normal object, as in the query shown in Figure 5, which asks for all the factories where commuters work.

![Figure 5: Using a Propositional View](image)

Figure 5: Using a Propositional View

Recursive views have to be defined in two steps: (1) the non-recursive base case is defined first. After this, the new class is immediately available in the query editor. Then (2) the recursive cases can be defined by adding further layers. Negation may be used in recursive rules, but the entire set of rules must remain stratifiable in a sense similar to the stratification of Datalog. We do not further elaborate on stratification and other prerequisites of admissible rules in this paper.

The definition of object generating views causes some problems in the presence of recursion. As noted above, we use a derivation process similar to the bottom-up evaluation of Datalog (Simply think of an object’s identity as an additional data-valued attribute taken from some special domain). Now, let us take a look at the set of rules commonly used to derive a path:

$$\text{path}(\text{start}=A, \text{end}=B) \leftarrow$$

$$A, B: \text{node} \land E: \text{edge}(\text{start}=A, \text{end}=B).$$

$$\text{path}(\text{start}=A, \text{end}=C) \leftarrow A, B, C: \text{node} \land$$

$$P: \text{path}(\text{start}=A, \text{end}=B) \land E: \text{edge}(\text{start}=B, \text{end}=C).$$

If every object derived by these rules takes a new identity then an infinite number of path objects will be generated, because on every round of the bottom-up evaluation there will be at least one path object with a new identity. To avoid this, we modify the usual bottom-up derivation procedure for our calculus in the following way: (1) A new object is derived by an object generating rule if and only if there is not already an object in the same class that is identical in all slot values and port references. Object identities are not considered in this comparison. (2) A new object is derived by an object preserving rule if and only if there is not already an object in the same class with the same identity. Slot values and port references are not considered in this comparison. Derivations according to this schema are guaranteed to terminate, because every variable used for the comparison to other objects can only be bound to finitely many different values.

5 Spatial Deduction

Let us turn to the deduction of spatial objects now. The techniques introduced so far are already sufficient for a very simple form of spatial deduction. We can, of course, derive objects with spatial attributes in the very same way as we have derived non-spatial attributes. Such an object can then be used in a query sketch just like any other object with a spatial attribute.

Assume, e.g., the class river has a single spatial attribute run of type polyline. If we want to define a subclass of river containing all the rivers which cross the border between Germany and some other country we can simply use the sketch given in Figure 3 as the view condition for a new class briver and assign to briver all of the attributes of river including the slot run. (Here it would be sensible to use an object preserving definition that assigns to each briver the identity of the corresponding river). If we now want to find the names of all towns lying by such a river the new view briver can be used in a query sketch like Figure 6. (Note the usage of the data-valued spatial function max_dist).

An interesting and important property of spatial views is that they can be used to encapsulate spatial conditions.
If, e.g., we formulate the query in Figure 6 directly, without using a view, we must use at least two different query conditions in separate layers, as the towns may lie in either country (Figure 7). Thus, views are a second way of expressing spatial “don’t care” conditions and an important means of simplifying complex queries.

Layer 1

The problem with this way of spatial deduction is that we can only derive spatial attributes whose values are already completely defined by some other attribute (as in the case above, where briver.run inherits the value of river.run). Things become more complicated if we want to calculate the value of a new attribute from two or more other attributes. Let us assume we had defined a view absentee with the query condition given in Figure 1. Let us further assume we wanted to define, say, a slot way: polyline for this class that contains the entire way an absentee has to drive from his house to his place of work. How can this value be derived from the three single spatial attributes of the two roads and the bridge? This is the point where spatial transformers (see Section 3) come into play: a transformer converts one or two objects into a new object and displays this object with a derived spatial attribute calculated on the basis of the arguments’ spatial attributes used in the sketch. Intersect, e.g., can be used to refer to the common area of two extended objects, merge can be used to refer to the union of two extended objects’ areas. The following example will illustrate the use of transformers and at the same time it will be the first example of a recursive pictorial view definition. It defines the fact that the continuous territories of a political union are formed by uniting adjoining territories of members of the union. The continuous territories of a union make up the derived class territory. We presuppose the existence of a simple object class union with the single slot name: string, of a class country with the geometric slot extension: pgon, and of a link class membership with the ports member: country and organization: union. The view definition consists of two layers. Layer 1 defines the trivial base case: each territory of a union member is the union’s territory (Figure 8). The attributes that have to be specified for the derived class territory are the spatial slot extension: pgon, whose value is the extension of the country, the non-spatial slot name: string, which is assigned the name of the union, and the port union: union, which links a territory to a union.

Layer 1

Figure 6: Usage of a Spatial View

Figure 7: A Multi-Layer Query

The translation of both layers of the pictorial view definition into DeLOM rules is:

\[
\begin{align*}
territory & (name=N, extension=E, union=U) \leftarrow \\
& C: country(extension=E) \cap U: union(name=N) \land \\
& M: membership(organization=U, member=C) .
\end{align*}
\]

\[
\begin{align*}
territory & (name=N, extension=A, union=U) \leftarrow \\
& C: country \cap U: union(name=N) \land \\
& M: membership(organization=U, member=C) \land \\
& T: territory(union=U) \land T neighbouring C \land \\
& X: merge(C, T) \land X(area=A) .
\end{align*}
\]

The following two queries both use this recursive view. A simple example given in Figure 10 employs the view to
search for the names of all towns with more than half a million of inhabitants inside of the European Community:

![Diagram of a town with population 500,000 inside a territory labeled "EC".]

**Figure 10: Using the View Territories**

To: town(pop=X) ∧ Te: territory(name="EC") ∧ To inside Te ∧ X > 500000.

The query in Figure 11 tests, whether some waterway is entirely located inside the territory of a single political union. Its interpretation is:

\[ W: \text{waterway} ∧ T: \text{territory} ∧ W \text{ inside } T. \]

![Diagram of a waterway inside a territory labeled "EC".]

**Figure 11: Spatial Encapsulation by a View**

Note that this query could not have been expressed on the basis of individual countries, as the waterway need not be located entirely within a single country even if it is located entirely within a single union.

### 6 Spatial Embeddings of Views

One thing remains to be done: the integration of propositional and spatial deduction. An object with spatial attributes can, of course, always be used in a propositional query graph as well. But how can we find a reasonable way to use purely propositional objects in sketches? There are, in fact, a lot of situations when such a possibility is desirable. Often objects which are characterized by spatial properties in the real world will only be represented by propositional knowledge in an information system, because no concrete spatial data is available or because we are not really interested in the exact spatial properties. Nevertheless, the conceptual view of these objects remains a spatial one. We have given an example for this in Section 1: connectedness of objects. Other examples can easily be found: imagine a systems that stores information about which airport belongs to which country in propositional form. There is no spatial information on the location available. Nevertheless, we would like to depict an airport in a sketch as a point-shaped object, because our intuitive concept is that it must be inside of the country which it belongs to. Another point is that non-spatial deduction has an important advantage over spatial deduction: it is computationally less expensive to base a derivation on propositional information than to calculate it from spatial information, because in a spatial derivation expensive geometrical tests have to be evaluated. Thus, it would be nice to be able to only pretend spatial deduction whenever exact spatial properties may be ignored or whenever we have already computed certain spatial properties before and have stored them in propositional form.

We will now show a way to use purely propositional information in spatial queries as if it were real spatial information. In the examples we presuppose, among other classes, the existence of a propositional view flight that models airlight connections. It has the ports from: city, to: city and a slot airline: string. If there is a link class in the extensional database that represents direct flights between two cities this view can be defined transitively in the same way as the view path in Section 4. Assume we wanted to plan a journey through Europe by plane and we wanted to see some European capitals on our way. In order to find an appropriate airline, we have to find out who is offering round-trips between at least, say, three European capitals. This query is given in Figure 12.

![Diagram of a network with flights between cities labeled as capitals.]

**Figure 12: Usage of an Embedded View**

The intended translation of the pictorial query is:

\[
\text{query(name=A) \leftarrow } \\
\text{Ca1, Ca2, Ca3: capital \land Ci1, Ci2, Ci3: city \land } \\
\text{F1, F2, F3: flight(airline=A) \land } \\
\text{T1: territory(name="Europe") \land } \\
\text{Ci1 inside T1 \land Ci2 inside T1 \land Ci3 inside T1 \land } \\
\text{F1(from=Ca1, to=Ca2) \land F2(from=Ca2, to=Ca3) \land } \\
\text{F3(from=Ca3, to=Ca1) \land } \\
\text{Ca1 is Ci1 \land Ca2 is Ci2 \land Ca3 is Ci3.}
\]

But since flight has no spatial attributes, we cannot use it in a sketch at all. To make this possible, we allow to define “faked” attributes for a view. These are spatial attributes that do not really exist and have no associated value. Nevertheless, they can be used to represent an object in a sketch. Thus, if we add a faked attribute route: line to the view flight we can sketch the query as shown in Figure 12. Still, with the standard rules of interpretation, the translation of this query is not what we intended (Differences printed in boldface):

\[
\text{query(name=A) \leftarrow }
\]
such a query could not be evaluated, because, e.g., the data for $F1.route$ that we would need to evaluate ($F1.touches C1$) does not exist. We therefore have to change the interpretation of objects with faked attributes. We cannot change the interpretation in a predefined way uniformly for all these objects, because we cannot know what a faked attribute is actually intended to mean. Thus, we have to provide a way for the user (or the database administrator) to define the reinterpretation of faked attributes. The method used for this is the definition of a spatio-propositional equivalence. Such an equivalence consists of (1) a spatial sketch that defines the way in which we want to use the faked attribute of some class and (2) a query graph that defines the appropriate translation of this usage. A faked attribute in combination with an equivalence definition is called a spatial embedding of a propositional view. In Figure 13 an appropriate embedding for $flight$ is given. The interpretation of the sketch is:

$$C1, C2; city \land F1:flight \land F1.touches C1 \land F1.touches C2$$

and the interpretation of the equivalent graph is:

$$C1, C2; city \land F1:flight(from=C2, to=C1)$$

![Figure 13: Spatial Embedding of the View Flight](image)

This definition means that every time we use a flight in a sketch to connect two cities, we want this to be interpreted as the corresponding link between these cities. With this embedding, the query shown in Figure 12 does result exactly in the intended translation given above. Thus, with spatially embedded views we obtain a way to work with pseudo-spatial attributes.

Formally, an embedding can be defined in the following way. Let $E$ be an embedding definition whose spatial part is given by a sketch $S$ and whose propositional part is given by a graph $G$. Let $I(S)$ and $I(G)$ be the interpretation of $S$ and $G$, respectively. Let $\Psi$ be the body of a query or view translation according to the standard rules of interpretation and let $\Phi = \phi_1, ..., \phi_k, ..., \phi_n$ w.l.o.g. be an appropriate reordering of $\Psi$. If $\Phi$ contains subformulas $\phi_k, ..., \phi_n$ that are identical to $I(S)^\theta$ except for an unambiguous renaming $\theta$ of variables ($\phi_1, ..., \phi_n = I(S)^\theta$) then these subformulas will be replaced by $I(S)^\theta$, i.e., $\Psi$ is transformed in the following way: ($\Psi \rightarrow \phi_1, ..., \phi_k-1, I(S)^\theta$) until no further embedding can be applied. All spatial constraints for faked attributes that remain after all possible transformations have been applied are removed.

View embeddings are a very powerful concept for the integration of spatial and propositional deduction, but have to be used with a certain amount of care, as ill-defined embeddings can result in confusing interpretations. Hence, embeddings should only be made available to an administrator, but not to an end-user.

7 Concluding Remarks

We have demonstrated how a departure from the classical map manipulation metaphor opens the way for pictorial deduction in spatial information systems. This observation illustrates a general matter in the design of information system interfaces: well-known metaphors taken from the domain of non computer-based work can be exploited to a certain degree to make a system readily accessible and comprehensible. Nevertheless, restricting the design too much by the limits of traditional patterns of work may prevent us from exploiting the genuine new opportunities that computer-based work offers.

We have presented the pictorial language Sketch that smoothly integrates the deduction of propositional and spatial data. This language can be regarded as a fully-fledged pictorial logic database language. In [10] we have shown that Sketch is at least equivalent to the class of stratified linear Datalog queries, i.e., it is more than relational complete. The basic metaphor of Sketch is sketching spatial scenarios. Stemming from the repertoire of common everyday communication skills, this metaphor offers a very natural way of spatial information access. We do, however, not deny that cartographic manipulation techniques are a central way of working with spatial information and it would be an interesting and important task to integrate our approach with map manipulation techniques.

Sketch has been implemented in Lisp (CLOS) for Unix systems running X11. An important issue that has to be tackled is to evaluate the acceptance of our approach by average database users.

References

[5] A. Frank. Beyond query languages for geographic databases: data cubes and maps in Geographic Database Man-


