in: Proc. 1st Int. Conf. on Deductive and Object-Oriented Database Systems,
Kyoto, Dec. 1989, pp. 535-554

Integrating Complex Objects and Recursion

Harald Schöning

University Kaiserslautern
P.O.-Box 3049
6750 Kaiserslautern
Federal Republic of Germany

The molecule-atom data model (MAD model) supports the management of complex objects in databases. It allows for the dynamic definition of complex object structures at query time. Since these structures may be recursive, transitive closure computations are supported in a natural way. The results of these computations do not only contain the elements belonging to a closure: structured representations of the transitive closure graph showing the path used to reach each node can be derived, too. Furthermore, path problems can be solved by appropriate operator combinations. The integration of the notion of complex objects with recursion makes the MAD model appropriate for many enhanced applications such as VLSI design, CAD/CAM and deduction.

1. Introduction

In recent years, many efforts have been made to integrate recursion into database management systems (DBMS). One main direction was the development of deductive DBMS which use recursion for reasoning and knowledge representation purposes (e.g. [1]). Less attention was paid to the need of recursion in DBMS for so-called enhanced applications (e.g. CAD/CAM and VLSI-design). These applications typically operate on complex objects (in contrast to the flat objects of classical database applications). Such complex objects may have a tree-like or a network-like structure often defined by means of recursion. Examples are "slicing trees" in VLSI design (describing a so-called *floorplan* by recursively partitioning the chip surface) and parts explosion in CAD. Furthermore, a version concept can naturally be modeled recursively [2].

As a consequence, DBMS developed for enhanced applications should support the notion of complex objects to enable an adequate modeling of the application objects within the DBMS. Operations in these systems should handle entire complex objects, allowing for structure-oriented projection and selection even in the case of recursive object structures.

There are many proposals for algorithms handling recursion (e.g. in the form of a transitive closure computation) in DBMS (see [3] for an overview), but less attention has been paid to the integration of recursion into data models and query languages. Nevertheless, simple forms of recursion have been proposed for some query languages. QBE, for example, only knows recursion for binary relations [4]. Horn clause languages for DBMS allow the formulation of recursion [5, 6]. These languages, however, are tailored for use in deductive DBMS, but complex object handling is not considered by them. On the other hand, data models designed to support complex objects (e.g. NF^2 [7]) are often based on a static object structure. Therefore, extensions of these models allowing for recursion are not capable of a *structured* representation of recursive complex objects [8], i.e., there is a discrepancy in the handling of recursive and non-recursive complex objects. POSTGRES [9] supports transitive closure computations and does not require statically predefined structures for complex objects. However, it does not combine these two facilities to enable a structured representation of a recursively defined complex object. Another QUEL extension [10] integrates transitive closure operations restricted to direct recursion on hierarchical structures into the query language without supporting a structured representation of their results. The data model of [11] allows for direct recursion in so-called configurations consisting of a hierarchy of complex objects, thus permitting a simple transitive closure computation. However, it remains unclear, whether or not the result can be further processed with the data model's operations.

Although [4] states the need for facilities to compute path problems as well as transitive closures, most of the models presented up to now concentrate on transitive closures. In contrast to that, our model is capable of computing the transitive closure as well path problems. Furthermore, recursion is not limited to direct recursion. The molecule-atom model integrates the notion of complex objects with recursion, in that it supports the structured representation of recursively defined complex objects, and handles recursive and non-recursive complex objects in a uniform way.

In the next chapter, we formulate requirements for the integration of recursion into a DBMS manipulating complex objects. The third chapter presents basic concepts of the MAD model. The recursion handling facilities of this model are presented in chapters 4 and 5. Chapter 6 shows some examples of the wide range of applications that can be modeled with these facilities.

2. Recursion in a Complex Object Environment

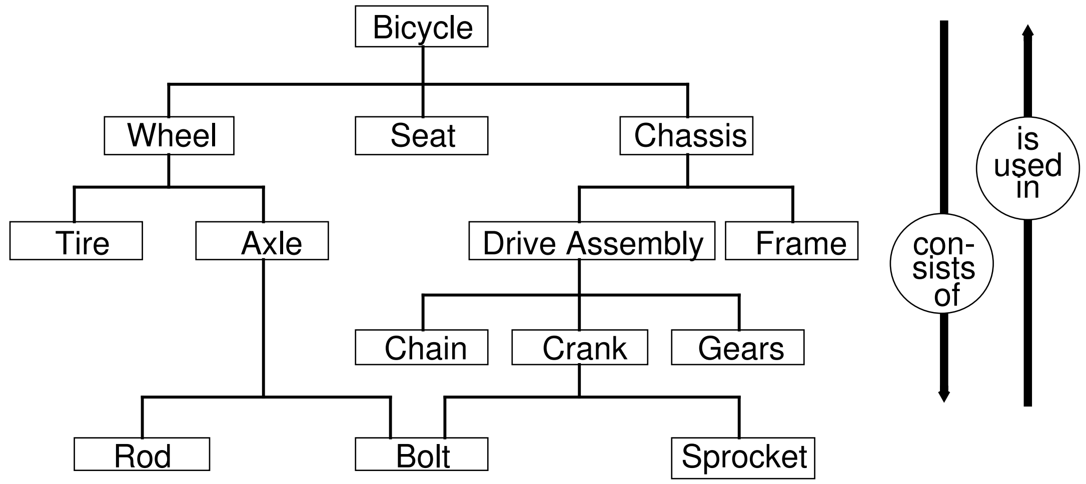
As mentioned above, complex object structures are often defined recursively. A machine, for example, consists of its parts, of their subparts, and so on. Thus, there is a relationship between parts. The "top-down" direction of this relationship is called *consists-of* and the "bottom-up" direction has the name *is-used-in*. Since the depth of a parts explosion is not known in advance, determining all parts (at all levels of the hierarchy) requires a transitive closure computation. In most proposals, the result of such a computation is an unstructured list of tuples, representing only one of the relationship's directions (as in figure 1b). This is sufficient to determine, whether a specific part is contained in the machine, but does not answer the question, where it is used and which subparts of the machine depend on this part. In other words, a **structured** representation of the transitive closure showing **both** the directions of the relationship is required (figure 1a).

Some algorithms proposed for transitive closure computations work on all data, i.e. compute the transitive closure of a whole relation. It is worth mentioning that in a complex object environment transitive closure operations commonly do not need to be computed over all data, but are performed selectively starting at a given root point ("partial transitive closure", in contrast to many recent discussions about transitive closure computations, e.g. [12]).

Another problem arising with recursive complex object structures is the computation of aggregate values from the objects' structures, i.e. not only to compute the transitive closure, but to derive values describing some properties of the transitive closure's structure. This kind of problem is commonly referred to as generalized transitive closure [4]. Regarding the transitive closure as a graph, the generalized transitive closure computation corresponds to the so-called path-problems [13]. This class of problems can be characterized by a combination of two operators: a *concatenator* combines the values within one path (consisting of several sections), and an *aggregator* computes the result from the values of all paths. Table 1 lists some path problems and the appropriate operator combinations. To compute the shortest path within a graph, for ex-

Application	Property	Aggregator	Concatenator
shortest path	length or time duration	min	+
critical (longest) path	length or time duration	max	+
maximal capacity path	capacity	max	minimum
most reliable path	reliability	max	*
bill of materials	item count	+	*
list all paths	edge name	$\cup$	concatenation
list any path	edge name	choose any	concatenation
propositional deduction	proposition	choose any	concatenation

Table 1: Some path problems and appertaining operator combinations [13, 4]



a) Structured view of a parts explosion (transitive closure of "consists-of") [10]

CONSISTS_OF

(MAJOR_PART, MINOR_PART)

(BICYCLE, WHEEL)
(BICYCLE, SEAT)
(BICYCLE, CHASSIS)
(WHEEL, TIRE)
(WHEEL, AXLE)
(AXLE, ROD)
(AXLE, BOLT)
(CHASSIS, DRIVE ASSEMBLY)
(CHASSIS, FRAME)
(DRIVE ASSEMBLY, CHAIN)
(DRIVE ASSEMBLY, CRANK)
(DRIVE ASSEMBLY, GEARS)
(CRANK, BOLT)
(CRANK, SPROCKET)

b) The CONSISTS_OF relation

additional tuples derived by a transitive closure computation

(BICYCLE, TIRE)
(BICYCLE, AXLE)
(BICYCLE, ROD)
(BICYCLE, BOLT)
(BICYCLE, DRIVE ASSEMBLY)
(BICYCLE, FRAME)
(BICYCLE, CHAIN)
(BICYCLE, CRANK)
(BICYCLE, GEARS)
(BICYCLE, SPROCKET)
(WHEEL, ROD)
(WHEEL, BOLT)
(CHASSIS, CHAIN)
(CHASSIS, CRANK)
(CHASSIS, GEARS)
(CHASSIS, BOLT)
(CHASSIS, SPROCKET)
(DRIVE ASSEMBLY, BOLT)
(DRIVE ASSEMBLY, SPROCKET)

Figure 1: Structured and unstructured (relational) view of a transitive closure

ample, one has to add the lengths of the sections within one path (concatenator +) and has to find the path with the minimal length (aggregator MIN).

On the other hand, a complex object is always defined by its components and their relationships to one another, even in the case of recursive object structures. Therefore, the recursion to be dealt with can be limited to the connections established by existing relationships among components (basic objects).

Up to this point, the notion of complex objects was used without further definition. For the following considerations, we define a complex object as a coherent, directed, and acyclic graph with a root. The nodes of this graph are basic objects, the edges are formed by one direction of the relationships among these basic objects. In figure 1, for example, the complex object (graph) is a *bicycle*'s parts explosion, the basic objects (nodes) are of type *part*, and the direction of the relationship forming the edges is *consists_of*. Alternatively, the *is_used_in* direction could have been used to construct a complex object with the semantics *depends on part*. The requirement for the graph to have a root and to be coherent and acyclic corresponds directly to the intuitive view of a complex object. Complex objects are not restricted to trees, but may be networks (as shown in figure 1). Thus, our definition is adequate.

It should be stressed that a transitive closure computation along one direction of a relationship with one starting point forms a complex object according to the above definition. E.g., all cities which can be reached from New York by railway form a complex object (assuming a relationship “has_railroad_connection_to" among “towns”). Statically predefined object structures are not sufficient in this case, since the depth of recursion can not be predicted.

In the next chapter, we present some basic features of the molecule-atom data model (MAD model), which supports dynamically constructed complex objects according to the above definition. The recursion handling facilities of this model are discussed in chapter 4 and 5.

3. Basic Concepts of the MAD Model

The MAD model [14] supports retrieval and manipulation operations on complex objects (called *molecules*). The structure of these molecules is defined at query-time, i.e., the objects are built dynamically at execution time rather than stored with a fixed structure. Input as well as output of the MAD model operations are sets of molecules, i.e., the result of a query consists of structured objects rather than of flat tuples as in the relational model.

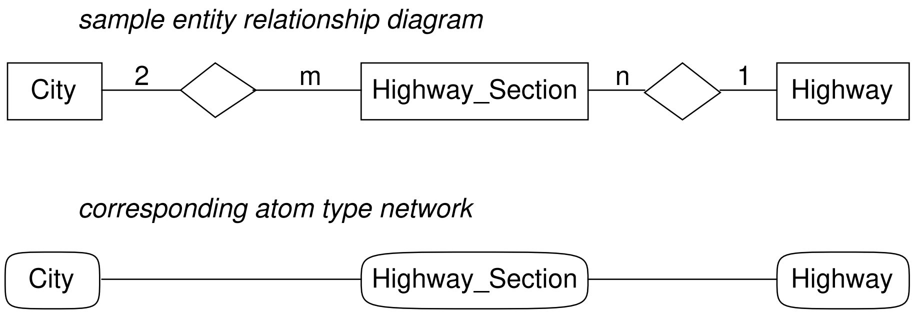
Molecules consist of atoms, which are comparable to tuples in the relational model, in that they consist of a set of attribute values and belong to one atom type (corresponding to a relation). Attribute values need not be defined for all attributes of an atom.

Each atom type has exactly one attribute of type IDENTIFIER which is used to uniquely identify the atoms (system defined surrogate). Binary attribute-free relationships of types 1:1, 1:n, and n:m can be directly expressed by attributes of type REFERENCE (repeating groups of IDENTIFIER values, in contrast to e.g. [15, where References are single-valued): If there is a relationship between atom type $\mathcal{A}$ and atom type $\mathcal{B}$, then $\mathcal{A}$ contains a REFERENCE attribute $\mathcal{A}$_to_$\mathcal{B}$, and $\mathcal{B}$ contains a REFERENCE attribute

$\mathcal{B}\_to\_\mathcal{A}$, which are pointing to one another. If there is a relationship between two atoms a (of type $\mathcal{A}$) and b (of type $\mathcal{B}$), $\mathcal{A}\_to\_\mathcal{B}$ of a contains the IDENTIFIER value of b, and $\mathcal{B}\_to\_\mathcal{A}$ of b contains the IDENTIFIER value of a. The values of the REFERENCE attribute pairs define edges of a graph whose nodes are atoms (*atom networ*k). Analogously, there is a corresponding undirected graph at the type level. The natural symmetry of real world relationships is automatically mapped to the MAD model schema, because each direction of the relationship is mapped to a REFERENCE attribute.

The example used to demonstrate the characteristics of the MAD model throughout the paper is chosen with respect to three requirements:

- The number of atom types involved should be small enough to make the examples attractive, but large enough to show relevant aspects.
- The semantics of the example must not require any special knowledge of a sophisticated application (such as CAD or VLSI design).
- The example's structure must be very general. Ideally, almost all applications should use structures which are special cases thereof.



attributes:

Id : IDENTIFIER	Id : IDENTIFIER	Id : IDENTIFIER
Name : String	Length : INTEGER	No : INTEGER
interstate_connections: REFERENCE TO (Highway_Section)	Exits: REFERENCE TO (City)	has_sections: REFERENCE TO (Highway_Section)
	belongs_to : REFERENCE TO (Highway)	

Figure 2: An entity relationship diagram and the corresponding MAD schema

For these reasons, we have chosen the highway example shown in figure 2 (type level), which represents non-hierarchical real world objects. Hierarchical structures are just special cases thereof. A simple entity relationship diagram modeling cities with their connections by sections of interstate highways as well as the corresponding MAD model schema are depicted. A highway obviously consists of a set of highway sections. We consider only a few attributes of the atom types (see figure 2). The IDENTIFIER attribute of each atom type is called *Id* for simplicity purposes; of course, other names are allowed. Cardinality restrictions could be added for each REFERENCE attribute, but have been omitted here. One aspect of the MAD model not represented by our example is a network-like type structure. We can neglect it here, since it does not influence the recursion handling facilities, which form the focal point of this paper.

Figure 3b shows a sample atom network for the schema of figure 2, representing some Texan cities and some interstate highways connecting them (figure 3a). The edges among the atoms represent the connections established between two atoms by the values of their REFERENCE attributes. For this reason, the values of REFERENCE attributes are not shown explicitly. Similarly, IDENTIFIER attributes are not listed. The atom representing San Angelo is not connected to any other atom, since there is no highway leading to San Angelo in our example.

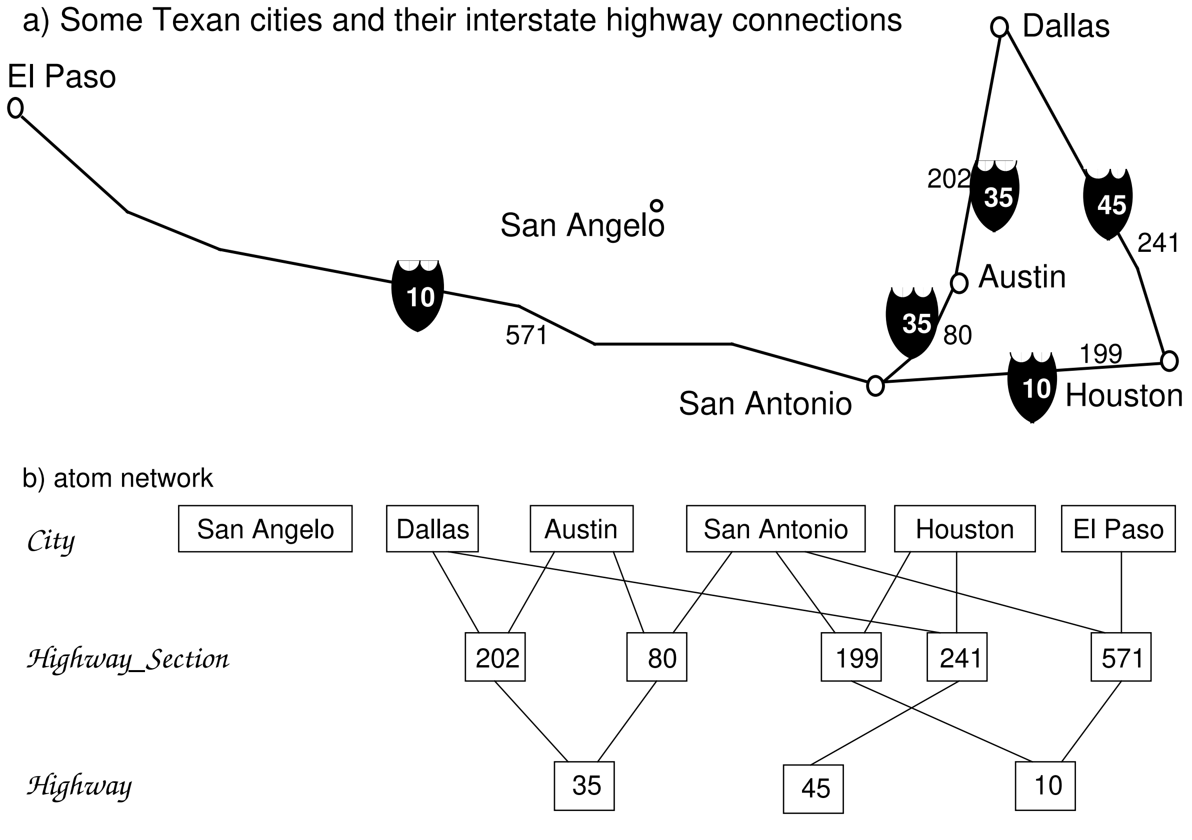


Figure 3: A sample map and its representation as atom network

The data manipulation language (DML) of the MAD model allows for the dynamic definition of molecule types based on atom type networks. For this purpose, atom types are selected, and a direction is given to edges connecting them, yielding a directed coherent graph with a root (the so-called *molecule type graph*). In a *molecule definition*, the directed relationships are represented by "-" preceded by the name of the REFERENCE attribute used, while the atom types are identified by their name. For example, the molecule type definition corresponding to figure 4 is *City.interstate_connections - Highway_Section*.

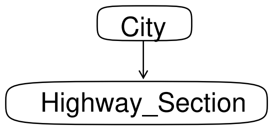


Figure 4: Molecule type graph for *City.interstate_connections - Highway_Section*

Since the direction of the relationship is specified at query time, *Highway_Section - City* is also a valid molecule type definition. If there is only one relationship between two atom types, the name of the corresponding REFERENCE attribute may be omitted. Thus, *City - Highway_Section* is sufficient in the above molecule type definition. Ramification is also possible, and is indicated by brackets and commas separating the branches. For example, *Highway_Section - (Highway, City)* corresponds to the molecule type graph of figure 5.

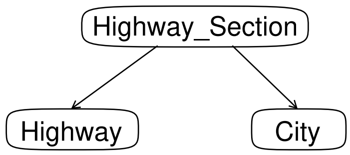


Figure 5: Molecule type graph for *Highway_Section - (Highway, City)*

An atom type may be included more than once in a molecule type graph. In this case, each occurrence of the atom type must be identified by a so-called role name. If, for example, the starting and ending points of a highway section are to be distinguished, one could define the molecule type *starting(City) - Highway_Section - ending(City)*.

In summary, molecules belonging to a molecule type can be described as follows:

- Each atom of the root atom type (*root atom*) induces one molecule.
- Atoms of other types belong to a molecule, if they can be reached transitively from its root atom using the specified directed relationships.
- Thus, a molecule is a coherent directed subgraph of the atom network starting with the root atom.

Hence, a molecule consists of a number of atoms together with the appropriate attributes and of structure information describing the molecule's shape. This structure information is depicted by arrows in a graphical representation of molecules. The molecule type definition *Highway - Highway_Section - City* applied to the atom network of figure 3, for example, yields the molecules shown in figure 6. Although the molecule type graph is a tree, the molecules have a network-like structure due to the fact that some highway sections share the same city.

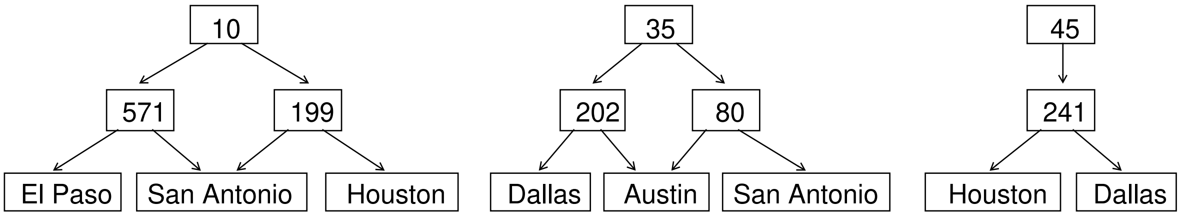


Figure 6: Molecules of type *Highway - Highway_Section - City*

In the following, we will concentrate on the description of retrieval statements, because they are sufficient to show all concepts involved in the following discussions. A retrieval statement (query) consists of a projection clause (**SELECT**), a molecule type definition clause (**FROM**), as described above, and a restriction clause (**WHERE**). The restriction clause contains conditions to restrict the set of molecules in the result. The projection clause specifies the atoms and the attributes of the atoms which are to be included in a result molecule. However, the coherency of the molecule must be preserved. The attributes to be projected can be specified in several ways:

- *atom type*: all attributes of this atom type are to be projected
- *atom type (list of attributes)*: the listed attributes are to be projected. The list may contain attribute names and so-called *virtual attributes* defined by assigning an expression.
- *atom type ()*: no attributes of this atom are projected

ALL is a short form for a list of all atom types appearing in the molecule type definition. *ALLBUT (list of atom types)* excludes the atom types listed from projection.

For explanation purposes, we introduce the following simple evaluation model for MAD model queries. Of course, this model does not correspond to an optimized implementation of molecule processing. The **FROM** clause is evaluated first, delivering all molecules of the specified type. Then the **WHERE** clause is used to restrict this set of molecules according to the condition specified. Finally, the structure of the molecules is modified as specified in the **SELECT** clause. The language is complete, i.e., a query is allowed wherever a molecule type definition is allowed, even within a molecule structure definition (**FROM** clause).

The question: *"How far are the next highway exits from San Antonio on the different highways?"*, for example, can be formulated as follows:

SELECT Highway_Section (Length), Highway (No), City (Name)
FROM Highway_Section - (Highway, City)
WHERE EXISTS City: City.Name = 'San Antonio'

and has the result set shown in figure 7. It is also possible to group the highway exits to one molecule instead of specifying a set of molecules as above, using the following query:

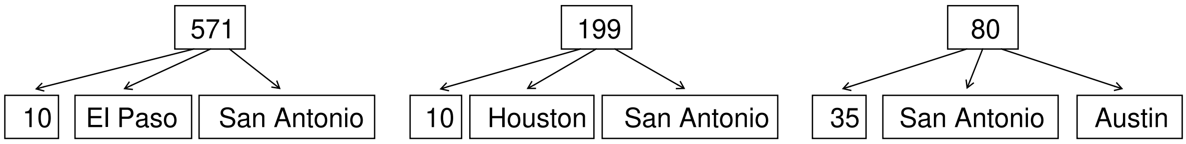


Figure 7: Distances to the next highway exits from San Antonio

```
SELECT  starting (Name), Highway_Section (Length),
        Highway (No), ending (Name)
FROM    starting (City) - Highway_Section - (Highway, ending (City))
WHERE   starting.Name = 'San Antonio'
```

The result set consists of one single element (figure 8). Here, the projection of *starting* is necessary to maintain molecule coherency. However, there is no need to project any attribute of starting. Thus, *starting()* is valid, too. The execution strategy is as follows: from San Antonio, three highway sections are found. From each of them, **all** cities indicated by the REFERENCE attribute *Exits* of *Highway_Sections* are included in the molecule, as well as the corresponding highway.

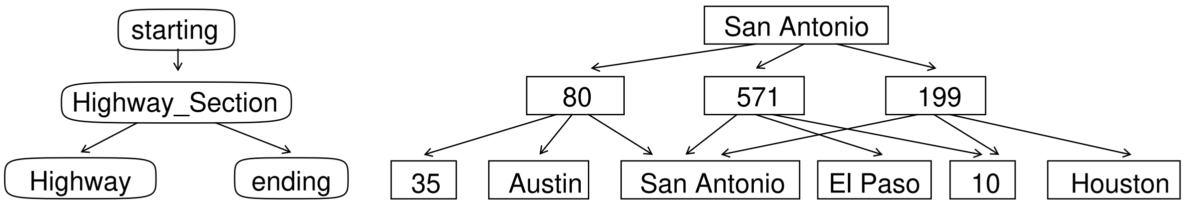


Figure 8: Distances to the next highway exits from San Antonio

Now, San Antonio appears once in role *starting* and once in role *ending*. This is not exactly the result one wants to see. The appearance of San Antonio in role *ending* can be suppressed by the so-called *qualified projection* which includes only those sub-molecules in the resulting molecules which fulfil the restriction clause. The keyword RESULT in the molecule type definition clause of the qualified projection indicates that the work is performed on submolecules of the molecules derived from the appertaining query. Atoms projected outside the qualified projection do not belong to the scope of the qualified projection. In the following query, which cuts off San Antonio in role *ending*, the scope of the qualified projection is the one-atom submolecule *ending*.

```
SELECT  starting (Name), Highway_Section (Length), Highway (No),
                 ( SELECT  ending (Name)                          ┐
                   FROM    RESULT                                 ├ qualified projection
                   WHERE   ending.Name <> 'San Antonio')          ┘
FROM    starting (City) - Highway_Section - (Highway, ending (City))
WHERE   starting.Name = 'San Antonio'
```

Qualified projection must, of course, maintain the coherency of the resulting molecules. Similarly, qualified projection could exclude the appearance of San Antonio in each of the molecules of figure 7.

In addition to the features presented previously, there are some built-in functions working on lists such as SUM, AVG, etc. The function VALUE converts set of molecules (each consisting of one atom with a single attribute) into a list. MOLAGG aggregates the value of an attribute over all atoms of one type within one molecule. The following example illustrates the use of MOLAGG and VALUE: *"Which is the highway connecting the highest number of cities and how long is it?"*

```
SELECT  Highway (No, Total := SUM (MOLAGG (Highway_Section.Length)))
FROM    Highway - Highway_Section - City
WHERE   COUNT (MOLAGG (City.Id)) = MAX (VALUE(
                  SELECT Number_of_cities := COUNT (MOLAGG (City.Id))
                  FROM   Highway - Highway_Section - City))
```

The query in the **WHERE** clause delivers one molecule consisting of a single atom with the attribute *Number_of_cities* for each highway in the database. *Number_of_cities* is an example of a virtual attribute, computed from values of the molecule's components which are aggregated to a list by MOLAGG. In order to be able to compute the maximum over this set of molecules, the result set has to be converted to a list by VALUE.

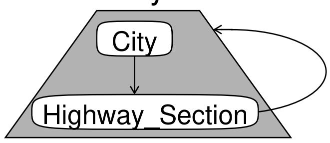
After this short introduction of some query facilities in the MAD model, we now concentrate on recursion handling in this data model.

4. Recursion in the MAD Model: Transitive Closure

The specification of recursion along a relationship is straightforward in the MAD model: One of the directed relationships used for molecule type definition is used to form a cycle in the type structure (indicated by the keyword RECURSIVE), thus allowing for an unlimited depth of recursion. The subgraph within this cycle is called *component molecule type*. Molecules of a recursive type are called *recursive molecules*. Such a molecule is derived from the atom network in analogy to non-recursive molecules: Starting from the root atom, the molecule is built up neglecting recursion (the resulting *root molecule* forms recursion level 0). Then, molecules of the component molecule type are appended to the root molecule as specified by the recursion forming directed relationship. In order to guarantee termination, a component molecule belongs to a recursive molecule exactly once: If a component molecule is already in the resulting molecule, it is not included again (cf. figure 9). This leads to the realization of a transitive closure computation with one starting point: A maximal directed acyclic subgraph of the atom network is formed (same approach as in [8] to avoid loops). The recursion levels can be addressed by indexing the names of the components. Furthermore, *LAST* and *ALLREC* can be used to reference the end of a recursion path or all recursion levels, respectively. *RECDEPTH* is a built-in function delivering the recursion level of its argument. In the highway example, recursion can be used to solve queries such as: *How can other cities be reached from San Antonio by highway?*

SELECT ALL
FROM City - Highway_Section RECURSIVE Highway_Section - City
WHERE City (0).Name = 'San Antonio'

molecule type graph



The recursion forming directed relationship is *Highway_Section - City*, the component molecule type is *City - Highway_Section*. The result set is shown in figure 9.

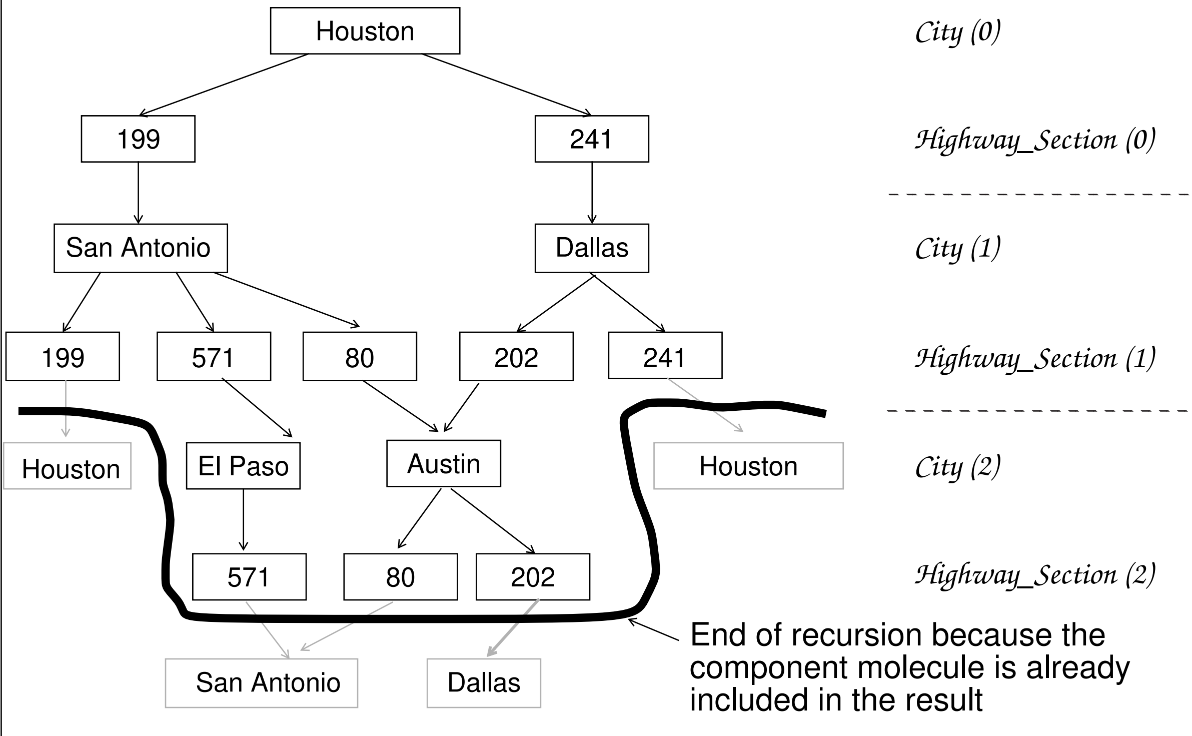


Figure 9: Transitive closure of *interstate_connections* starting with Houston

Since recursion termination is defined in terms of component molecules, the highway sections belonging to the cities in the recursive molecule are always included, even in the last recursion level.

It should be stressed that the result does not only contain the information as to which cities can be reached, but also, how they can be reached, i.e. a structured view of the transitive closure is derived rather than the flat view generated by many other approaches. Analogously to the starting point, defined in a very natural way in the MAD model approach, there may be the wish to specify stopping criteria for the recursion, e.g.

- limit the depth of the recursion or
- stop when a certain node is reached or
- stop when the paths used do not fulfil a certain condition.

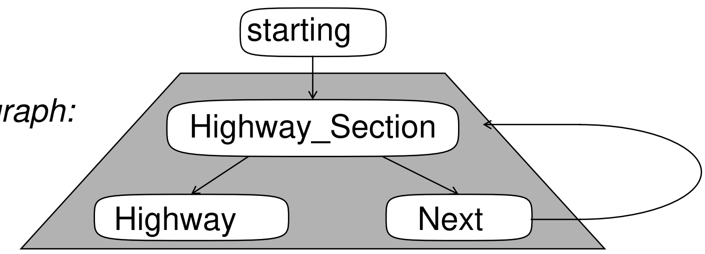
For these purposes, the UNTIL clause can be used, which has the following semantics: A component molecule is not appended to the recursive molecule, if the condition of the UNTIL clause turns out to be TRUE after its inclusion. For the formulation of conditions on a single component molecule, we introduce the keyword CURRENT to identify the component currently in doubt. The question: *"Which routes can be chosen from Dallas without touching San Antonio?"* can be expressed as follows:

```
SELECT  ALL
FROM    City - Highway_Section RECURSIVE Highway_Section - City
                  UNTIL City (CURRENT).Name = 'San Antonio'
WHERE   City(0).Name = 'Dallas'
```

To enable path-dependent conditions, the keyword PREVIOUS identifies the direct (already included) predecessor of the current component molecule as the following example shows: *"Which cities can be reached from El Paso without changing the highway?"*

```
SELECT  starting (Name), Highway_Section (ALLREC) (), Next (ALLREC) (Name)
FROM    starting (City) - Highway_Section - (Next (City), Highway)
                 RECURSIVE Next - Highway_Section
                 UNTIL Highway (CURRENT).No <>Highway (PREVIOUS).No
WHERE   starting.Name = 'El Paso'
```

corresponding molecule type graph:



Until now, we have discussed transitive closure computation in the MAD model. As mentioned in chapter 2, enhanced applications also require facilities to cope with path problems. Chapter 5 shows the embedding of constructs allowing for the handling of path problems in the MAD model.

5. Solving Path Problems with the MAD Model

The transitive closure representing molecules derived from queries with a RECURSIVE clause contains only minimal paths (concerning number of traversed relationships). These paths, however, are not sufficient for the computation of some path problems. For this reason, we introduce the REC_PATH clause which generates all (concerning number of traversed relationships) maximal cycle-free paths according to the specified directed relationship. This is achieved by following only one value of a REFERENCE attribute at a time, instead of following all of them in the case of the RECURSIVE clause. Each of the paths generated this way forms a molecule of its own. In order to continue the construction strategy of the path molecules, the atoms of a component molecule are included only if they directly belong to the recursive path or if their type does not contribute to recursion at all (like *Highway* in the previous example). This is the reason, why only one *Highway_Section* is present at each recursion level in the following example. The question "*Find all paths from Houston to El Paso*" illustrates the difference between the clauses RECURSIVE (cf. figure 9) and REC_PATH and yields the result shown in figure 10.

SELECT ALLBUT (Highway_Section (LAST))
FROM City - Highway_Section REC_PATH Highway_Section - City
WHERE City (0).Name = 'Houston' AND City (LAST).Name = 'El Paso'

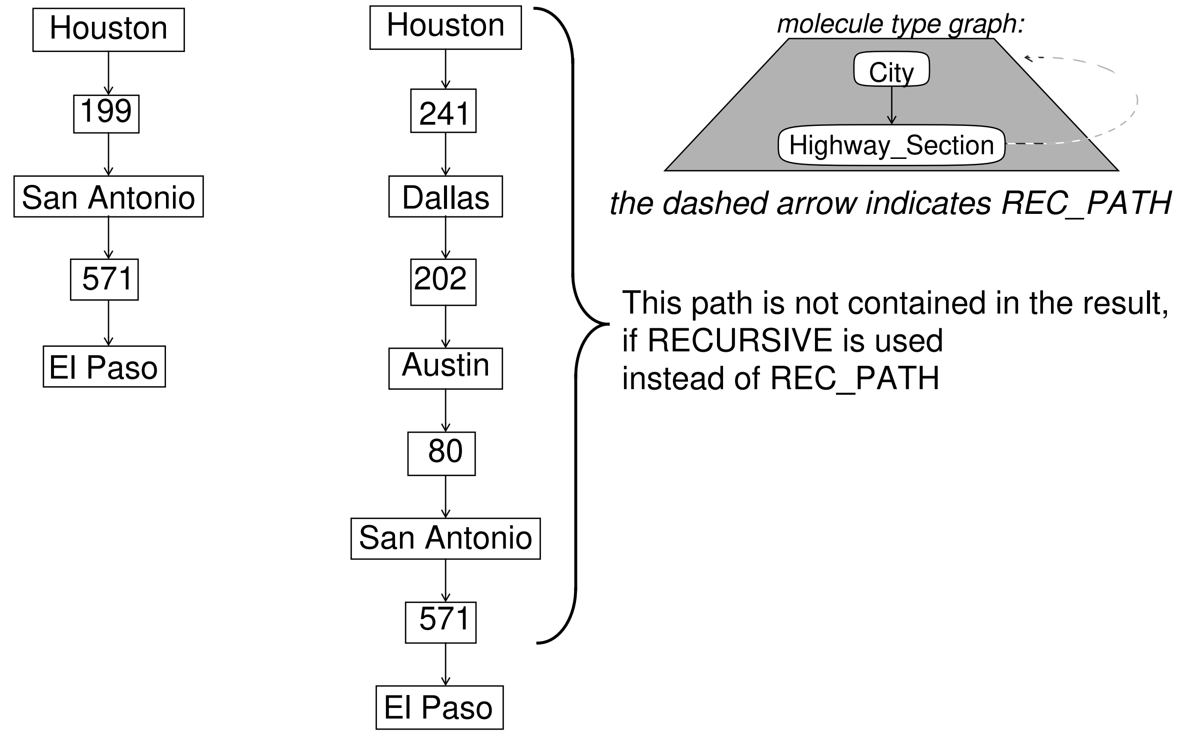


Figure 10: The result of the REC_PATH query

The molecules derived by REC_PATH can be directly used to compute path problems in combination with MOLAGG and VALUE. Table 2 shows the appropriate MAD model operators for aggregation and concatenation (cf. table 1).

Application	Property	Aggregator	Concatenator
shortest path	length/duration	MIN (VALUE...)	SUM (MOLAGG...)
critical (longest) path	length/duration	MAX (VALUE...)	SUM (MOLAGG...)
maximal capacity path	capacity	MAX (VALUE...)	MIN (MOLAGG...)
most reliable path	reliability	MAX (VALUE...)	MULT (MOLAGG...)
bill of materials	item count	SUM (VALUE...)	MULT (MOLAGG...)
list all paths	edge name		[MOLAGG]
list any path	edge name		[MOLAGG]
propositional deduction	proposition		[MOLAGG]

Table 2: MAD model operators for path problems

Since molecules are already structured, there is no need for a concatenator in the last three lines of table 2. A REC_PATH query without restriction clause yields all paths. Thus, an aggregator for “List all paths” is not necessary. Similarly, each molecule represents a path, making an explicit *choose any* operator superfluous. In these cases, MOLAGG can be used to “flatten” the structure of a molecule.

We give an example for the problem “shortest path”: *“How long is the shortest distance from San Antonio to Dallas?”*

```
SELECT starting (miles:=MIN(VALUE( SELECT pathlength
                                   FROM   RESULT         Aggregation
                                   WHERE  To = 'Dallas'))))
FROM  (SELECT starting (),
         New :=                                Concatenation
         (pathlength := SUM (MOLAGG (Highway_Section (ALLREC).Length)),
         To := ending (LAST).Name)
      FROM     starting (City) - Highway_Section - ending (City)
                      REC_PATH ending - Highway_Section
         UNTIL ending (PREVIOUS).Name = 'Dallas'
      WHERE   starting.Name = 'San Antonio' )
```

This query illustrates the completeness of the query language (cf. chapter 3). We discuss its semantics step by step: The molecules specified by the inner **FROM** and **WHERE** clauses are shown in figure 11a. The inner **SELECT** concatenates the lengths of the highway sections by adding them, thus yielding the total path length (as virtual attribute *pathlength*). Each value derived this way forms, together with the destination (*To*) of the corresponding path, a new atom. All new atoms are grouped to one molecule with the starting city as the root atom (figure 11b). The qualified projection within the aggregation selects the two submolecules with **To='Dallas'** from this molecule. Finally, the minimum of their pathlength values is computed.

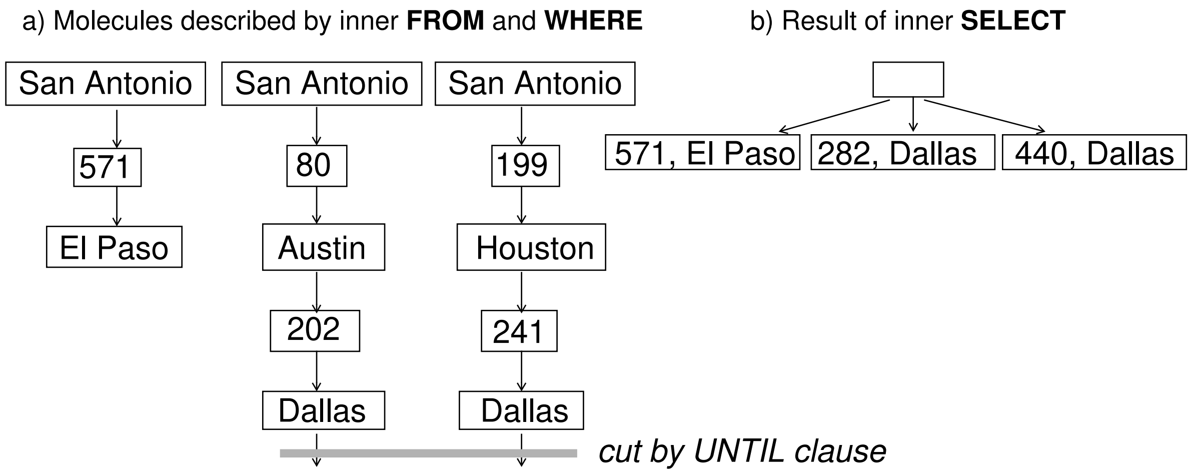


Figure 11: Illustration of the sample shortest path query

6. Further Comments on Functionality and Applications

The structure of a component molecule type is not limited, i.e., the component molecules themselves may be recursive. An application of this nested recursion facility is sketched in the following example: Imagine, you want to know how to reach all cities and not only those touched by an interstate highway. In this case, one has to include other roads into the model. Figure 12 shows a map with some additional roads (connecting San Angelo to the other cities) and one way to model this map in the MAD model.

The question: *"Show me ways from Houston to San Angelo starting on an interstate highway"* can be formulated as follows:

```
SELECT  ALL
FROM    starting (City) - Highway_Section - (Highway, ending (City) -
              Roads := (SELECT ALL
                        FROM    Road_Section - (C (City), Road)
                                  REC_PATH C - Road_Section
                                  UNTIL C (PREVIOUS).Name = 'San Angelo'
                        WHERE   C (LAST).Name = 'San Angelo')
              REC_PATH ending - Highway_Section
              UNTIL EXISTS Roads (PREVIOUS)
WHERE   starting.Name = 'Houston' AND EXISTS Roads (LAST)
```

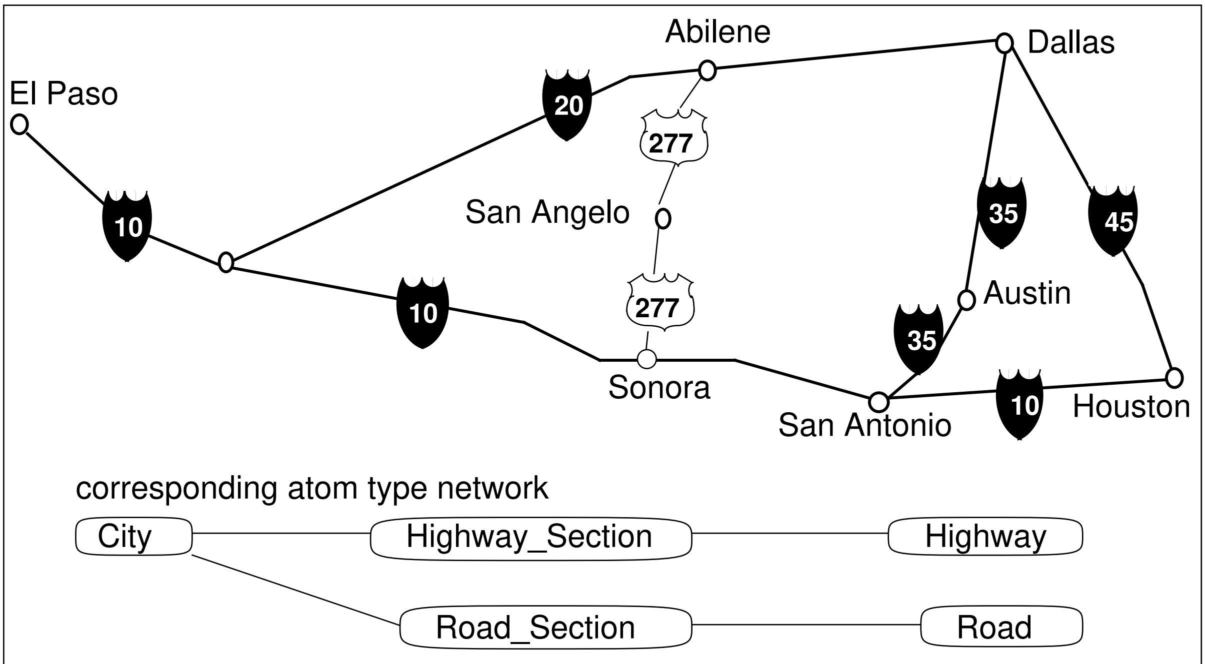
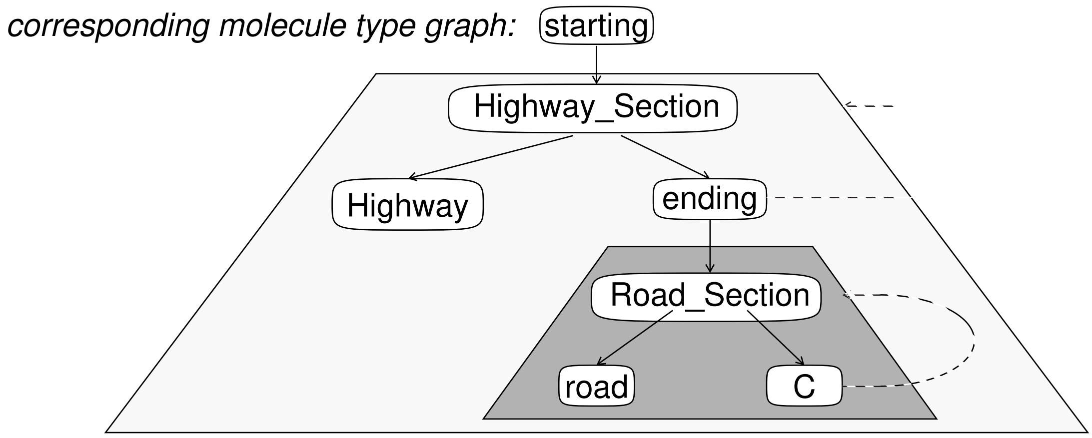


Figure 12: Adding other roads to the city/interstate highway map



The inner recursion delivers all connections leading from the current exit of the interstate highway to San Angelo by other roads.

Of course, it is interesting to consider how far the recursion handling facilities of the MAD model can be used for deductive databases. Deductive database systems based on the relational model commonly realize predicates as relations, and facts as tuples in this relation. In order to store the fact *“John is parent of Mary”,* for example, there must be a relation ***parent***, where a tuple ***(John, Mary)*** is stored. Right sides of rules combining predicates are transformed into joins over the appropriate relations. Recursion in these rules cannot be handled within the pure relational model, and therefore has to be realized on top of it by a program, or must be integrated as an extension of the relational model.

In the MAD model, predicates are expressed by relationships. If, for example, a predicate B is represented by a REFERENCE attribute $\mathcal{B}$, then the fact B(a, b) is represented by a relationship between the two atoms a and b via an appropriate value of a’s attribute *B*. Thus, the structure B(x, z), C(z, y) corresponds to *x.B - z.C - y*. The above example *"John is parent of Mary"* would be realized by a REFERENCE attribute *is_parent* within the atom type *Person* and *is_parent* of the atom representing John containing the IDENTIFIER value of the atom representing Mary .

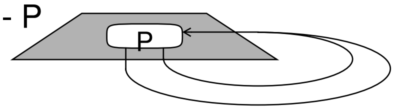
Recursive rules are mapped onto recursive molecule types. A set of rules representing the ancestor relationship

Ancestor (x, z)	:-	Parent(x, y), Ancestor(y, z)	more naturally expressed by :
Ancestor (x, y)	:-	Parent (x, y)	Ancestor (x,?) :- Parent $(x,?)^+$

corresponds to the molecule type definition *x.Parent - y* RECURSIVE *y.Parent - y*.

Deduction based on the MAD model is further supported by its facilities for dynamic schema modifications (allowing for the introduction of new predicates) and molecule type pre-definition. The further discussion of these topics, however, is beyond the scope of this paper. Here, we stress only two aspects: Predicates are automatically bidirectional in the MAD model: The definition of a predicate *is_parent* implies the definition of *has_parent*. As already mentioned, the natural symmetry of real world relationships is directly mapped into the model. Furthermore, multiple relationships among atom types may be united, as the following example illustrates: A distinction between *has_father* and *has_mother* does not impose problems for the computation of the ancestor relation. In the following query, the ancestor relation is given in a structured view:

```
SELECT ALL
FROM   P:=(SELECT Name, parents := has_father&has_mother
           FROM   Person) RECURSIVE P.parents - P
```



Thus, the MAD model also supports rule processing in deductive applications. Additionally, knowledge base management systems can be based on the MAD model as it was shown in [14].

7. Conclusions

The need for recursion is obvious in many database applications, particularly in enhanced applications such as CAD/CAM. Here, recursion is required in combination with the notion of complex objects. Using the MAD model as example, we have shown, how these two aspects can be integrated into a data model and a query language.

Transitive closure as well as path problems can be computed employing these facilities. In contrast to many other approaches, the structure of even recursive objects can be represented within the data model, i.e., recursive and non-recursive objects are handled in a uniform way. Thus, many applications where objects of this type are used (e.g. VLSI design and CAD) are supported by the MAD model. Furthermore, the MAD model can be used for deduction in a natural way.

We did not address questions of implementation, but concentrated on data model and query language aspects. The language constructs to express recursion may appear quite complex in the case of path problems. This is caused by the generality of the language. This complexity can be broken down by the introduction of specialized language constructs like "Shortest_Path (Dallas, San Antonio, City-Highway_Section)", which can be expanded automatically to the queries discussed above.

The RECURSIVE clause has been implemented in the PRIMA project [16, 17]; a second version of PRIMA also offering the REC_PATH clause is under implementation.

8. Acknowledgments

I would like to thank T. Härder and A. Sikeler for reading an earlier version of this manuscript and for helpful suggestions to improve the presentation, as well as I. Littler for preparing the manuscript.

9. References

[1] Bayer, R.: Database Technology for Expert Systems, in: Wissensbasierte Systeme, Informatik Fachberichte 112, Springer, pp. 1-16.

[2] Klahold, P., Schlageter, G., Wilkes, W.: A General Model for Version Management in Databases, in: Proc. 12. Int. Conf. on VLDB, Kyoto, 1986, pp. 319-327.

[3] Bancilhon, F., Ramakrishnan, R.: An Amateur's Introduction to Recursive Query Processing Strategies, in: Proc. ACM SIGMOD, Washington, DC, 1986, pp. 16-52.

[4] Dayal, U., Smith, J.M.: PROBE: A Knowledge-Oriented Database Management System, in: Brodie, M.L., Mylopoulos, J. (eds.): On Knowledge Base Management Systems, Springer, 1986, pp 227-257.

[5] Raschid, L., Su, S.Y.W.: A Parallel Processing Strategy for Evaluating Recursive Queries, in: Proc. 12. Int. Conf. on VLDB, Kyoto, 1986, pp. 412-419.

[6] Ullman: Implementation of Logical Query Languages for Databases, in: ACM TODS, Vol. 10, No. 3, 1985, pp. 289-321.

[7] Scheck, H.J., Scholl, M.H.: The Relational Model with Relation-Valued Attributes, in: Information Systems, Vol. 11, No. 2, 1986, pp. 137-147.

[8] Linnemann, V.: Non First Normal Form Relations and Recursive Queries: An SQL Based Approach, in: Proc. 3rd IEEE Int. Conf. on Data Engineering, Los Angeles, Feb. 1987, pp. 591-598.

[9] Stonebraker, M., Rowe, L.A.: The Design of POSTGRES, in: Proc. ACM SIGMOD, Washington, DC, 1986, pp. 340-355.

[10] Tillquist, J., Kuo, F.-Y.: An Approach to the Recursive Retrieval Problem in the Relational Database, in: CACM, Vol. 32, No. 2, 1989, pp. 239-245.

[11] Kim, W., Chou, H.-T., Banerjee, J.: Operations and Implementation of Complex Objects, in: Proc. 3rd IEEE Int. Conf. on Data Engineering, Los Angeles, Feb. 1987, pp. 626-633.

[12] Agrawal, R., Jagadish, H.V.: Direct Algorithms for Computing the Transitive Closure of Database Relations, in: Proc. 13. Int. Conf. on VLDB, Brighton, 1987, pp. 255-266

[13] Carré, B.: Graphs and Networks, Clarendon Press, Oxford, 1979.

[14] Mitschang, B.: Towards a Unified View of Design Data and Knowledge Representation, in: Proc. 2nd Int. Conf. on Expert Database Systems (EDS), Tysons Corner, Virginia, 1988, pp. 30-50.

[15] Lorie, R.; Plouffle,W.: Complex Objects and Their Use in Design Transactions, in: Proc. Data Base Week, San Jose, 1983, pp. 115-121.

[16] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. 13. Int. Conf. on VLDB, Brighton, 1987, pp. 433-442.

[17] Härder, T. (ed.): The PRIMA Project - Design and Implementation of a Non-Standard Database System, Research Report 26/88, SFB 124, University Kaiserslautern, 1988.