OPERATION-ORIENTED QUERY LANGUAGE APPROACH FOR RECURSIVE QUERIES: PART 1 FUNCTIONAL DEFINITION

Timo Niemi, University of Tampere, Department of Computer Science, P.O.Box 607, SF-33101 TAMPERE, Finland.
Kalervo Järvelin, University of Tampere, Department of Library and Information Science, P.O.Box 607, SF-33101 TAMPERE, Finland.

Abstract
So far the aspects related to efficient processing have dominated the research on recursive queries. In this paper we consider how the formulation of recursive queries can be made easier from the view point of the non-professional user - also in the context of complex recursive queries. It is obvious that the conventional rule-based way of defining is too hard and cumbersome for many non-professional users. We provide operations at a high abstraction level in terms of which the user can formulate his recursive queries in a compact and convenient way. In our approach recursive processing is needed for constructing transitive relationships among data. In practice, it is often very important to compute transitive relationships among several union-compatible binary relations instead of one binary relation as usual. We define the operations so that they are able to manipulate transitive relationships among several relations. For the changing needs of the user our approach contains three kinds of operations: relation-oriented, node-oriented and path-oriented operations. In this paper we specify a functional language consisting of operations of these types and give several examples on how the user can formulate his recursive queries in terms of this language. We also discuss its role in deductive databases, i.e. its integration with processing based on an extensional database.

Keywords: Deductive databases, recursive queries, transitive relationships, knowledge representation, functional specification.

1. Introduction

It is typical of DBMS's that they are capable of offering for users two kinds of information: explicitly stored and derived. Traditionally, the derived information is produced from the explicitly stored data through the view mechanism. However, advanced applications (e.g. many CAD/CAM and information retrieval applications) have revealed their incapability of defining and processing structurally complex objects. Especially it has been recognized that the view mechanisms are insufficient in producing all necessary derived information. The greatest disadvantage is that they do not allow views which presuppose recursive definition and processing. For example, it is widely known that it is impossible to produce virtual relations or views which require construction of the transitive closure of a binary relation with the relational algebra [1]. This is because the definition of the transitive closure is recursive.

During the last years considerable attention has been paid to how the recursive expressive power can be integrated into the conventional databases. The possibility of recursive definition is so essential from the view point of the user that it is one of the basic starting points in modern object-oriented database approaches (see e.g. [2], [3]). Logic is an effective tool both for expressing recursion and for defining the essential issues related to databases.

For example, Kowalski [4] has shown that logic is a uniform language for defining facts, rules, programs, queries, views and integrity constraints. In other words, we can define all derived information - also in recursive cases in terms of logic.

Recursive rule definitions have a central role in deductive databases. Deductive databases or logic(al) databases are databases in which the derived information is defined by logic-based rules (called the IDB or Intensional Data Base) from the explicitly stored data (called the EDB or Extensional Data Base). Usually an IDB is represented by Horn clauses. Very often only so called Datalog programs (see e.g. [5]) are allowed. Datalog programs consist of Horn clauses without function symbols, i.e. they allow only variables and constants as arguments of predicates.

The development of efficient evaluation methods has dominated research related to recursive queries. Some evaluation methods have been intended for general recursive queries whereas others have been tailored to some useful subsets of recursive queries. It is a very hard task to define and classify recursion and its different types comprehensively and generally [6]. Therefore it is also difficult to define generality exactly in the context of evaluation methods. The evaluation methods intended for general recursive queries can be divided into two categories: actual evaluation methods and rewriting methods. Actual evaluation methods themselves are sufficient to produce the answers for recursive queries whereas the rewriting methods presuppose that they are followed by some actual evaluation method. In fact rewriting methods are optimization techniques which transform rules so that recursive queries can be performed more efficiently after transformation. Well-known actual evaluation methods are e.g. naive evaluation [7], semi-naive evaluation [7], the Henschen-Nagvi method [8], Prolog (see e.g. [9]) and rewriting methods are e.g. magic sets [10] and counting [11]. Many general evaluation methods have been introduced and compared widely in [12].

When algorithms are developed for subsets of recursive queries it is very important to find such limited recursion types which are sufficient for the needs of most applications. It has been accepted widely that the linear recursive queries are the most common recursion class occuring in practice. We say that the definition of a predicate is linearly recursive if the recursive predicate appears once and only once at the right side of a rule. Linear recursive definitions correspond to the Chang's concept 'regularity' [13]. It has been shown in [14] that each linear recursive query can be expressed in terms of such an operation sequence in which the transitive closure operation is possibly preceded and followed by conventional relational operations. This means that the transitive closure has a dominant role among linear recursive queries.

Due to the great practical value of the transitive closure many authors propose that it should be incorporated into the expressive power of conventional query languages. For example, its incorporation into QBE [15], into SQL [16], [17] and into Quel [18] has been proposed. Agrawal has introduced the so-called α-operator in terms of which the capability of manipulating the transitive closure can be added to relational algebra [19]. The transitive closure operation is also included in the so-called traversal recursion approach [20]. There are also several algorithms (e.g. [21], [22], [23], [24], [25], [26], [27]) which have been developed for implementing the transitive closure. We will also concentrate in this paper on examining transitive relationships among data.

Aspects related to efficient computation have dominated the research on recursive queries so far. In this paper we rather investigate how the definition of different transitive relationships

could be made easier from the view point of a non-professonial user. From the view point of the user we can divide the existing approaches into three main categories: rule-oriented, graph-oriented and operation-oriented approaches. The most usual approach to recursive queries has been that the user formulates a recursive query by using logic-based rules. A typical work related to this approach has been introduced e.g. in [28] where the integration of rules with relational databases has also been considered.

It is obvious that in the rule-oriented approach the user has to master recursion in the conceptual sense very well and he has to have a capability of strong recursive thinking. The definition of any recursive rule presupposes that the user is able to formulate generally a rule for transferring from one level of recursion to another and, in addition, to give the exit rule for terminating the recursion. In complex cases the definitions of many recursive rules can be associated with each other. It is obvious that in practice the construction of recursive rules, at least in complex cases, is too heavy for most non-professionals.

Moreover, existing systems developed for recursive queries require that the user masters in detail the underlying evaluation strategy. For example, the user can define the transitive closure in the logical sense correctly as follows. First he formulates the recursive rule so that the definition is left recursive and second he gives the exit rule. However if we in this case would use e.g. Prolog as our evaluation strategy then the computation would never terminate due to the evaluation strategy of Prolog. We think that the non-professional user has to be able to express his recursive queries without knowing aspects related to the evaluation strategy. It is the duty of the logic programmer to know the underlying evaluation strategy but it is not the duty of the non-professional user. Gruz et al. [29] have recognized the difficulties associated with formulations of recursive rules and they introduce a graphical query language in terms of which recursive queries can be formulated in a simpler way.

Rosenthal et al. [20] have introduced an interesting graph-oriented approach to recursion called traversal recursion. It is based on the traversal of a graph. In traversal recursion a graph consists of nodes and edges so that each node and edge contains some associated information expressed with node and edge labels, respectively. It is typical that the output of a query based on traversal recursion is another graph which consists of nodes and edges derived from the original graph. The node and edge labels of the derived graph are constructed when the original graph is traversed. A file structure, which efficiently supports traversal recursion in large acyclic graphs, has been developed in [30] .

Agrawal's approach [19] is an example of an operation-oriented approach. The α-operator in this approach allows the definition of the transitive closure between two attributes with the same domain. Both the argument and the result of the α-operator are relations. This means that the α-operator is a relational operation in the formal sense. In turn this means that the α-operator can be freely intermixed with other relational operations. The relational algebra, which contains the traditional relational operations and the α-operator, is called the α-extended relational algebra. In Agrawal's approach linear recursive queries are expressed as a sequence of relational operations belonging to the α-extended relational algebra. The α-operator in this algebra affords the possibility of defining transitive relationships among data.

It is desirable that the user can formulate his queries in one homogeneous way. In the context of non-procedural languages, non-professional users have practiced in defining their information needs by combining available operations with each other. Therefore we choose

the operation-oriented approach in this paper. Because also Agrawal's approach is operation-oriented we consider differences between the starting points of these approaches.

Data, on which transitive computation is based, is organized in a different way. Agrawal's [19] idea was to develop an advanced mechanism for transitive computation which seamlessly works with other relational operations. The α-operator performs transitive computation among the so-called distinquished attributes of its operand relation. In our approach to deductive systems we connect our mechanism intended for transitive computation to the EDB which can be based on different data models. In this paper we consider principles in terms of which this mechanism can be connected to the EDB. In Part II [31] we deal with the situation that the EDB is based on the relational model.

We think that in many applications from the user view point it is most appropriate to group the data intended for recursive computation into a collection of binary relations instead of one binary relation. This is due to the fact that the principle used for sharing contains essential information in the semantic sense. We call an individual binary relation an ERP-relation (an extensional basic relation intended for recursive processing) and all ERP-relations constitute an ERP-base. This means that we have to search transitive relationships among several ERP-relations when performing recursive queries. We will demonstrate that the capability of our approach to computate transitive relationships in a certain scope (in a collection of ERP-relations) reduces effort of the user in query formulation.

In Agrawal's approach all transitive computation, also aggregation related to it, is performed by one powerful α-operator. However, it is typical of many applications, e.g. CAD/CAM applications that the user is interested only in data which are in a specified transitive relationship with given data. Especially the direction of transitive computation is often important. For instance, in this paper our example deals with part hierarchies. By specifying the direction of transitive computation we can express whether we are interested in those products that include the given part or those components which are included in this part. In order to support query formulation in such cases more specific operations than one transitive closure operation is needed. The language defined in this paper offers such operations. In addition, our language contains non-transitive operations such as top_nodes and bottom_nodes (see 4.2) which find data in special positions from a collection of ERP-relations. These operations are also useful in many applications (see e.g. [32]).

In addition to part-component hierarchies we have applied our approach to information retrieval [32]). We apply it to classifications of terms on different abstraction levels. Transitive computation is needed to find out the superclasses and subclasses of given terms. Our operations have been defined so that the data in the ERP-base are acyclic whereas Agrawal's α-operator allows also transitive computation among cyclic data. This means that our approach cannot, without extensions, be used e.g. for processing road networks.

Also the integration principle of transitive computation with conventional database computation is different. Agrawal's approach is based on the extension of the relation model such that the extension is consistent with other relational processing. In our approach this integration happens via standard operation(s) of the EDB. This means that in our approach we have to tailor integration for each alternative data model which can be used in the organization of the EDB. On the other hand we do not want to restrict our approach to any data model which must be used in the organization of the EDB. In Part II [31] we give a prototype implementation for integration in the case that the ERB-base is based on the relational model. The integration with other data models is in progress.

From the view point of an existing relational DBMS Agrawal's approach means that the new α-operator has to be implemented and integrated with other relational processing. Because Agrawal's α-operator extends the set of the relational operations it means a.o. that the optimization techniques developed for relational expressions have to be extended to expressions involving the α-operator [19]. The so-called heterogeneous way for implementing a deductive database system is based on an interface which is implemented between two separate systems, i.e. we have a deductive system and a DBMS. In this implementation way it is desirable that the execution of recursive queries can be performed without modifying radically the architecture on which the conventional dbms processing is based [33]. Our starting point is that some EDB-based operation(s) can be used in the integration. In Part II [31] we show that a standard operation (restriction operation) can be used in the integration when the EDB has been organized according to the relational model. This affords the possibility of using the functional language defined in this paper as a deductive system also in the context of a heterogeneous implementation way. We discuss more this issue in Chapter 6 of Part II and demonstrate how conventional optimization techniques of relational databases, without any modification, can be used in the optimization of expressions consisting of ERP-based and relational operations.

In Agrawal's approach [19] the so-called additional attribute Δ contains the derivation history of the transitive computation. It is typical of the attribute Δ that its scope is only the α-operator and no relational operator (except for projection) are allowed on Δ outside the α-operator. As Agrawal [19] points out, this means that the α-operator cannot be used to express a query that requires intersection of two or more traversal histories. In other words the common elements of two or more transitive computations cannot be found. However this would be a useful feature in many CAD/CAM and IR applications. Therefore our functional language contains operations which support finding of common elements related to different transitive computations. We will give examples on these operations.

Our functional language consists of operations at a high abstraction level. These operations can be divided into three categories: relation-oriented, node-oriented and path-oriented operations. Next we consider briefly features characteristic of different operation types included in our language. In fact we apply in this paper only one relation-oriented operation (∑_rel_union-operation). This operation identifies exactly those ERP-relations among which transitive relationships are considered. The only role of this operation is to specify the ERP's from the ERP-base or the scope of transitive computation. The scope has to be given for each node-oriented or path-oriented operation. Thus this operation is embedded into the definitions of all other operations of our language. By restricting our consideration to only specific ERP-relations we can express essential semantic information as we shall see later in our examples.

Each ERP-relation or any collection of ERP-relations can be visualized as graph. An element in an ERP-relation or in a collection of ERP-relations corresponds to one node in the graph visualization. The node-oriented operations are used to examine which elements (nodes) are in certain positions in an ERP-relation (or in a collection of ERP-relations) or which elements are in certain transitive relations with the given elements. The first operations are typically non-recursive in nature whereas the latter ones are recursive. We offer for the non-professional user the node-oriented operations at high abstraction level. In our approach, recursive manipulation is invisible from the view point of the user in cases which require the construction of transitive relationships among elements. Our aim is to develop such operations that the user can formulate his queries on the basis of concepts which have the

natural correspondences in the universe of discourse at hand. For example, we provide operations for the common successor or predecessor nodes of a given node set.

Sometimes the user is not satisfied with knowing only what elements have the desired transitive relationship with given elements. The user is often interested also in possible paths to connect the specific elements with each other in a collection of ERP-relations. We define path-oriented operations for this purpose. Our aggregation operation involving transitive computation is associated with path-oriented operations because paths retain those elements (nodes) which are needed in the transitive computation. Due to the fact that our aggregation operation needs also information from the EDB we consider this operation separately. The operations of our functional language are based purely on the ERP-base whereas the aggregation operation needs the integration of ERP-based and EDB-based processing. In this paper we consider both EDB-dependent and EDB-independent aspects of our aggregation operation.

It is very important to all database applications that they are defined exactly and generally. Generality means that the definition is not bound to any sample case and preciseness guarantees that the application has a clear and unique semantics (meaning). If the meaning of the application is unclear then it is difficult to understand, assess and use the application. This is why the VLDB panel of Mexico City on "Type Specification and Data Bases" proposed in its summary report [34] that in the future, a new model, feature, or language should not be accepted without a precise and exhaustive specification. Therefore we also define our approach and language with its operations exactly in this paper. It is worth noting that the formal definition of our approach leaves many alternatives to implementing it. The notational conventions, which we need in the formal definition, is defined in Chapter 2.

In Chapter 3 we consider how we represent the extensional information intended for transitive computation in our approach. In other words we pay attention to how the data in ERP-relations and in an ERP-base are organized and what structural constraints among the data must hold. In this chapter we also give a sample ERP-base which we use to demonstrate our approach through the whole paper. The different relation-oriented, node-oriented and path-oriented operations are introduced and defined exactly in Chapter 4. The functional language consisting of these operations is defined in Chapter 5. In Chapter 6 we demonstrate how the user can formulate his queries in a straightforward way with this language - also in those cases which require complex recursive manipulation. These examples show that the recursive manipulation is invisible to the user. In this chapter we also discuss how, in terms of the operation-oriented approach, we can avoid several troublesome aspects which are related to a recursive rule-based query formulation from the view point of the user. Our approach to deductive databases means that the functional language defined in this paper has to be connected to some EDB. In Chapter 7 we deal with this issue. Also aggregation related to transitive computation presupposes the integration of the ERP-based and EDB-based information. Therefore we introduce also our aggregation principle in this chapter. Thus Chapter 7 is the bridge to Part II [31].

2. Basic Notations

Our formalism in this paper is based on the following notational conventions.
Notational convention1: The *power set* of a set S is denoted by P(S). For example, if S = {a,b,c} the P(S) = {{},{a},{b},{c},{a,b},{a,c},{b,c},{a,b,c}}.

Notational convention2: The *finite n-tuples* (briefly *tuples*) are denoted between angle brackets, for example <a,b,c>. The symbol <> means the *empty tuple*. We use also this notational convention for tuples in our dse (data structure element) formalism (see e.g. [35], [36]).

Notational convention3: The *tuple set* of a set S is denoted by T(S). The elements in T(S) are tuples whose components belong to the set S. The set T(S) is formed so that it contains each permutation, represented as a tuple, for each element belonging to the set P(S). For example, if
S = {a,b,c} then T(S) =
{<>,<a>,<b>,<c>,<a,b>,<b,a>,<a,c>,<c,a>,<b,c>,<c,b>,<a,b,c>,<a,c,b>,<b,a,c>,
<b,c,a>,<c,a,b>,<c,b,a>}. It is obvious that |T(S)| > |P(S)| if |S| ≥ 2 (the notation |A| means the cardinality of the set A).

Notational convention4: The *length* of a tuple t is the number of the elements in the tuple and it is denoted length(t). The length of the empty tuple is 0. For example, length(<a,b,c>) = 3.

Notational convention5: The symbol □ is used to refer to an element in a tuple. For example,
a □ <a,b,c> is true.

Notational convention6: The *signature* of a function is denoted by f:D∅R where f is a *function symbol*, D is a *domain set* i.e. it defines a set of values to which the function can be applied and R is a *range set* i.e. it defines a set of values which contains the results of function applications.

3. The Representation of Extensional Information for Recursive Processing

In the database area two essential abstraction levels are distinquished: the schema and instance levels. In the shema level we describe the organization of data and the instance level contains the actual data which conform to the given schema. Recursive processing is associated purely with the instance level. Very often the need for the recursive processing occurs among objects of the same type. It is characteristic of recursive processing that we are intrested in the transitive relationships among objects whereas in the non-recursive processing of databases different properties related to objects are interesting. In other words the integration of non-recursive processing with recursive processing is an essential question. We consider this integration in [31]. In this paper we concentrate on recursive processing.

3.1. The Formal Representation for Extensional Informat

It is desirable that the objects and relationships among them are represented in a compact manner for recursive processing. Therefore we describe for recursive processing explicitly only the identifiers of objects and the immediate relationships of objects on the basis of these identifiers. In this paper the set, from which the identifiers of objects can take their values, is denoted by *Nodes*. The definition of the set depends entirely on the universe of discourse. As we shall see the set Nodes contains elements which are nodes when our representation is interpreted graphically.

Definition1: If a and b are two object identifiers (i.e. a□Nodes b□Nodes) and the *immediate relationship* among of these objects exists then the immediate relationship is denoted by the tuple <a,b>.

It is important to note that the order in the immediate relationship is essential i.e. <a,b>≠<b,a>. In the context of any immediate relationship <a,b> we use the following terminology. We call the element a an *immediate predecessor* of the element b and the element b an *immediate successor* of the element a. It is typical that the immediate relationship expresses the basic structure which has its own semantic meaning (see our example below).

One of the ideas of deductive databases is to allow the processing of the general predecessors and successors of objects. By grouping immediate relationships we get structures which contain implicitly also other predecessors and successors than the immediate ones. For example, if we have two immediate relationships <a,b> and <b,c> then a is an indirect predecessor of c and c is an indirect successor of a. We believe that in practice it is often most appropriate to group the immediate relationships into separate groups because the grouping principle contains essential information in the semantic sense. We shall also see later on that by grouping immediate relationships into separate groups it is possible to process only relevant immediate relationships from the view point of a query instead of all immediate relationships. This has of course a positive effect on the efficency of processing.

Definition2: We call a set achieved by grouping immediate relationships on the basis of the used grouping principle an *extensional basic relation intended for recursive processing* or briefly an *ERP-relation*.

Formally, this means that an ERP-relation is a binary relation i.e. it is a subset of the Cartesian product Nodes ∞ Nodes. We will refer to each constructed ERP-relation by its unique name. In other words if A is an ERP-relation then A can be represented explicitly as follows: A = {<a1,b1>,..., <an,bn>} where a1, ... , an,b1, ... ,bn □ Nodes.

Definition3: An *ERP-base* is a collection of ERP-relations. Formally, An ERP-base is the following set {erp-rel$_1$,erp-rel$_2$, ... ,erp-rel$_n$} where each erp-rel$_i$ (i□{1, ... ,n}) is a subset on Nodes ∞ Nodes.

Because each ERP-relation in an ERP-base is a subset on Nodes ∞ Nodes it means that all ERP-relations are union-compatible with each other, interpreted in terms of the terminology of the traditional relational model. In practice the construction of an ERP-base from the universe of discourse resembles the process of discovering union-compatible relations in the context of the traditional relational databases. Formally, it is obvious that an ERP-base and any of its subset is an element in the set P(P(Nodes ∞ Nodes)) and an ERP-relation is an element in the set P(Nodes ∞ Nodes).

3.2. The Sample ERP-base

Our example is associated with the following production process. We have a company which makes products in three cities Munich, Stuttgart and Berlin. It can use these products for the following purposes:
1) It can make from these products new products in the same factory.
2) It can make from these products new products in the factory of another city.

3) It can sell these products.
It is important to note that the nature of this production process is recursive, because the company constructs further products from products made by itself. This kind of a production process binds some products very closely with each other. Likewise the decision making in this kind of a production process is much more complicated than in the traditional one. Let's imagin that we have to make a decision on the quantity of a certain product that must be made. It is not possible to make the decision solely on the basis of the market situation of this product as usually. We must also to take into account the market situation of those products in which the product at hand is an immediate or indirect component.

As we mentioned above we represent relationships among objects by grouping the immediate relationships into separate ERP-relations. Objects in our example are products and we associate the identifiers p1,p2, ... ,p19 with our sample products. The natural grouping principle of immediate relationships is the factory or city where the production expressed by an immediate relationship occurs. This grouping principle means that our sample ERP-base contains three ERP-relations named by Munich, Stuttgart and Berlin. In our example the immediate relationship <p2,p1> is interpreted so that the product p1 is used immediately in the construction of the product p2 and the formal expression <p2,p1> □ Munich means that the immediate construction of the product p2 by using the product p1 as one component in it occurs in Munich. In other words the ERP-relations themselves contain essential information. In Figure 1 we represent explicitly our sample ERP-base. We will use this sample ERP-base throughout this paper.

Sample-ERP-base = {Munich,Stuttgart,Berlin}

Munich = {<p1,p10>, <p1,p11>, <p1,p12>, <p2,p1>, <p2,p13>, <p3,p1>, <p3,p10>, <p3,p11>, <

Stuttgart = { <p7,p1>, <p7,p8>, <p14,p7>, <p14,p15>}

Berlin = {<p9,p16>, <p9,p17>, <p9,p18>, <p8,p17>, <p8,p19>}

Fig. 1. The sample ERP-base.

It is worth noting that our sample ERP-base in Fig. 1. can be represented explicitly as follows {{<p1,p10>, ... ,<p6,p13>},{<p7,p1>, ... ,<p14,p15>}, {<p9,p16>, ... ,<p8,p19>}}. This representation exemplifies that our sample ERP-base is an element in the set P(P(Nodes ∞ Nodes)). We can represent also the binary relations belonging to our sample ERP-base graphically. In Figures 2a, 2b and 2c we represent the graphs related to the ERP-relations, respectively. Likewise any combinaton of the ERP-relations corresponds to a graph. In Figure 2d we give a common graphical representation for all sample ERP-relations.

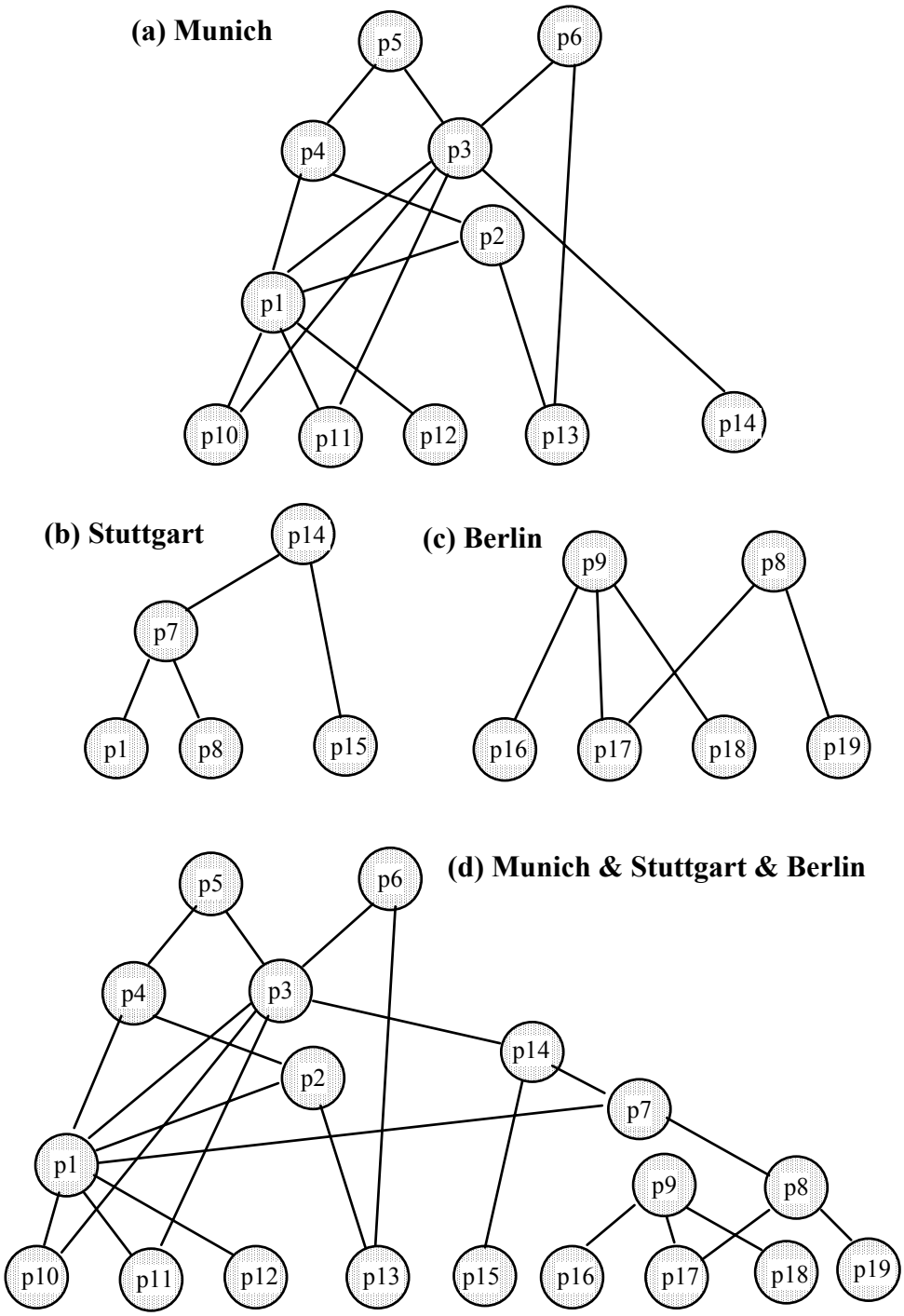

Fig. 2a-d. The graph visualizations for the ERP-relations Munich, Stuttgart, Berlin and the common graph visualization for all sample ERP-relations.

We can find two kinds of elements which have a special position in ERP-relations. In any ERP-relation there are some elements which have no predecessors or no successors. We call the former *top nodes* and the latter *bottom nodes*. It is typical that the top nodes and the bottom nodes have their own semantic meaning. In our example the top nodes identify that there are no production processes after their construction. We call this kind of products end products of production. In our example bottom nodes identify those products from which the production process in this factory starts. We call these products starting products. In principle the starting products, in our example, have two different sources: they can be bought from

suppliers or they are transported from other factories producing them. For example, the ERP-relation Munich has the following top nodes or end products {p5,p6} and the following bottom nodes or starting products {p10,p11,p12,p13,p14}. From these starting products the product p14 has been made in Stuttgart and the other products have been bought from outside of the company. It is worth noting that any collection of ERP-relations has its own top nodes and bottom nodes. For example, the bottom nodes of the collection {Munich,Stuttgart} are p10, p11, p12, p13, p8 and p5.

Consider an example on how complex the relationships among objects in our sample ERP-base can be. We consider in this example the products p1 and p3 and how they are associated with each other. The product p1 is essential for the production of the product p3 in two different ways. The first one is that the product p1 is one of those four products which is needed as an immediate component in the constructing of the product p3 in Munich. The second one is that the product p1 is implicitly in the product p14 which in turn is also one of the immediate components needed in the construction of the product p3. The construction of the product p14 requires the transportation of the product p1 from Munich to Stuttgart (because p1 is one bottom node in the ERP-relation Stuttgart). The construction of the product p14 happens in Stuttgart (in fact the product p14 is the only end product made in Stuttgart). And finally the product p14 has to be transported to Munich for constructing the product p3 (p14 is one of the bottom nodes in the ERP-relation Munich).

3.3. The Acyclicity Constraint Related to the Represen

We manipulate in this paper only binary relations which graphically correspond to acyclic directed graphs. In practice we have many situations that presuppose acyclicity among nodes. For example, if we would allow in our sample ERP-base cycles among nodes it would mean that a product could be indirectly a component of itself. Of course this is an impossible situation. Therefore we require that any representation for extensional information have to satisfy the acyclicity constraint. The functional definition of the acyclicity constraint is based on the transitive closure of the binary relation. Next we define the transitive closure on the basis of our formalism.

Let R and S be any two binary relations on Nodes ∞ Nodes. The *composition* of R and S is denoted by R∘S and it is defined as follows.

Definition4:

$\circ$: P(Nodes ∞ Nodes) ∞ P(Nodes ∞ Nodes) ∅ P(Nodes ∞ Nodes)

R∘S = {<a,c>|∃b□Nodes:<a,b>□R3<b,c>□S}

On the basis of the composition we define the n^{th} power of a binary relation R and we denote it by *power(R,n)*. Intuitively power(R,n) gives those node pairs whose distance from each other in the corresponding graph visualization is n edges provided n is less or equal than the height of R (see Definition 7). The power of a binary relation R is defined as follows when we denote the set of positive integers by I^+.

Definition5:

power: P(Nodes ∞ Nodes) ∞ I^+∅ P(Nodes ∞ Nodes)

power(R,n) =

R ,if n=1

R∘power(R,n-1) ,if n>1

For example, related to our sample ERP-base power(Munich,4) = {<p5,p10>,<p5,p11>,<p5,p12>}.

Intuitively, the transitive closure of a binary relation contains all immediate and indirect relationships among of objects of a given binary relation. Next we define the transitive closure of a binary relation.

Definition6:
transitive_closure: P(Nodes ∞ Nodes) ∅ P(Nodes ∞ Nodes)
transitive_closure(R) = ≈$_{i\square\{1, \ldots ,\infty\}}$ power(R,i)

For example, related to our sample ERP-base transitive_closure(Stuttgart) = {<p7,p1>, <p7,p8>, <p14,p7>, <p14,p15>, <p14,p1>, <p14,p8>}. Intuitively in the construction of transitive_closure(R) the binary relation power(R,k+1) contains one edge longer indirect relationships among objects than the binary relation power(R,k) does. It is obvious if we have a finite binary relation R then there exists a positive integer i such that power(R,i) ≠ {} and power(R,j) = {} always when j>i. This positive integer is called the height of the binary relation (or sometimes the depth of the transitive closure [24]). Intuitively, if i is the height of a binary relation then power(R,i) expresses the longest indirect relationships which can be found in this relation.

The height of a binary relation is defined formally as follows.

Definition7:
height: P(Nodes ∞ Nodes) ∅ I^+
height(R) = n; n□I^+: power(R,n)≠□ 3 power(R,n+1)= □

For example, height(Munich) = 4 because power(Munich,4) ≠ □ and power(Munich,5) = □. In the context of finite binary relations it is interesting to compute powers for the transitive closure only until the height of a binary relation because after that it is not possible to produce new indirect relationships.

Intuitively, a binary relation contains a cycle if we can find from this binary relation two elements (nodes in the graphical visualization) a and b such that on the other hand the immediate relationship <a,b> exists and on the other hand the indirect relationship from the element b to the element a can be derived in terms of the powers of the original binary relation. Next we give a formal definition for the cyclicity and acyclicity of a binary relation.

Definition8: A binary relation R on Nodes ∞ Nodes is *cyclic* if ∃b,a□Nodes 3 ∃n□I^+:<a,b>□R 3 <b,a>□power(R,n). Otherwise a binary relation is *acyclic*.

As we explained above all binary relations included in an ERP-base have to be acyclic. Next we define the function acyclic_checking which checks whether a given binary relation (the argument of the function) is acyclic or not. The function requires that we compute powers one after another. We construct power in the order power(R,1), power(R,2), ... and for each power we check if the cyclicity condition specified above holds. If we can reach

power(R,height(R)) without satisfying the above cyclicity condition then the binary relation R is acyclic. In the definition we refer to set {true,false} by Boolean.

Definition9:
acyclic_checking: P(Nodes ∞ Nodes) ∅ Boolean
acyclic_checking(ERP-relation) =
through_powers(ERP-relation,ERP-relation)
where through_powers is defined recursively as follows

through_powers: P(Nodes ∞ Nodes) ∞ P(Nodes ∞ Nodes) ∅ Boolean
through_powers(Power, ERP-relation) =
false, if ∃ <y,x>□ERP-relation 3 ∃ <x,y>□Power
true, if Power = □
through-powers(Power∘ERP-relation, ERP-relation), otherwise

For example, related to our sample ERP-base the function acyclic_checking(Munich) yields the value true.

We considered above the possible complexity of relationships among the elements in different ERP-relations of an ERP-base. Therefore we have to require that an ERP-base at hand does not contain any cycle. We can define this kind of a constraint related to an ERP-base by constructing one common binary relation representation from all ERP-relations of an ERP-base and by checking that this representation does not contain a cycle. We give the function *acyclic_ERP_base* for this purpose.

Definition10:
acyclic_ERP_base: P(P(Nodes ∞ Nodes)) ∅ Boolean
acyclic_ERP_base(ERP-base) = acyclic_c
where common_representation is defined as follows
common_representation: P(P(Nodes ∞ Nodes)) ∅ P(Nodes ∞ Nodes)
common_representation(ERP-base) = $\approx_{i \square ERP\text{-}base} i$.

For instance related to our example acyclic_ERP_base(Sample-ERP-base) yields the value true. This means that our sample ERP-base has an acyclic representation.

4. Operations for Manipulating an ERP-base

In principle there are three kinds of information in an ERP-base which may be of interest to the user: ERP-relations themselves, elements (nodes) in an ERP-base among of which certain relationships hold and paths between elements in an ERP-base. Therefore we need for these purposes three operation categories which can be characterized (*binary) relation-oriented* operations, *node-oriented* operations and *path-oriented* operations, respectively. Each operation category contains its own operations.

Because ERP-relations themselves contain essential semantic information it is natural that the user specifies in his query those ERP-relations which are relevant to him. For instance, related to our example the user may be intrested only in the part of the production process which happens either in Stuttgart or Berlin. This is expressed in terms of our relation-oriented operation.

The result of node-oriented operations consists always of elements included in the ERP-base at hand i.e. it consists of nodes if we interpret the result graphically. A path consists of a finite number of nodes and in addition the order between these nodes is important. Therefore we represent paths as tuples. The construction of paths is defined by path-oriented operations. Next we consider operations belonging to the different categories in detail.

4.1. (Binary) Relation-oriented Operations

Actually we apply in this paper only one binary relation-oriented operation which can be considered as the generalization of the union operation of the traditional relational model. Because binary relations in an ERP-base are union compatible with each other we can define the union of two binary relations R1 and R2 as follows.

Definition11:
rel_union: P(Nodes ∞ Nodes) ∞ P(Nodes ∞ Nodes)∅ P(Nodes ∞ Nodes)
rel_union(R1,R2) = {<x,y>|<x,y>□R1Δ<x,y>□R2}

The result relation of the above operation contains the immediate relationships of only two binary relations. We need such an operation in terms of which the user can choose any part of an ERP-base. Therefore we have to generalize the above operation to permit any number of ERP-relations. This is defined by the Σ_rel_union function.

Operation 1:
Σ_rel_union: P(P(Nodes ∞ Nodes)) ∅ P(Nodes ∞ Nodes)
Σ_rel_union({ERP-rel1,ERP-rel2, ... ,ERP-reln}) =
{<x,y>|<x,y>□ERP-rel1 Δ<x,y>□ERP-rel2,Δ ... ,Δ<x,y>□ERP-reln}

It is obvious that the argument of the function Σ_rel_union can contain only ERP-relations of the ERP-base at hand i.e. the function expression Σ_rel_union(X) has to satisfy the condition ERP-base □ X. For example, if in the context of our sample ERP-base the user is interested only in the part of the production process which happens either in Stuttgart or Berlin, he can express this by the function Σ_rel_union({Stuttgart,Berlin}) which gives the binary relation {<p7,p1>, <p7,p8>, <p14,p7>, <p14,p15>, <p9,p16>, <p9,p17>, <p9,p18>, <p8,p17>, <p8,p19>}. In other words, the result does not contain that part of the production process which happens in Munich.

It is also worth noting that the above formal definition for Σ_rel_union does not say anything about how we should implement this operation, for example it is not necessary to construct any result binary relation physically. In our prototype implementation [31] Σ_rel_union goes only through the immediate relationships of those binary relations which have been expressed in its argument. Related to our sample expression, we can interpret this kind of processing as follows: in the processing of the graph of Fig. 2d we omit irrelevant nodes from the view point of the user i.e. nodes related purely to Munich. From the efficency point of view it is important that we can process a relevant subgraph instead of the whole graph. If we want to process the whole sample graph in Fig. 2d or the whole sample ERP-base in Fig 1 we can do it by the expression Σ_rel_union({Munich,Stuttgart,Berlin}).

4.2. Node-oriented Operations.

In general, the idea of the node-oriented operations is to offer for the user expressions at a high abstraction level in terms of which he can get those elements which have certain properties or which are in a certain relations with some other elements. Because each element in an ERP-base corresponds to one node in its graphical counterpart we call these operations node-oriented operations. Typically the operations, which are used to get elements with certain properties in an ERP-base, are defined non-recursively, whereas operations intended for producing elements, which are in a certain transitive relation with given elements, are

usually defined recursively. In Chapter 5 we will define a functional language which consists of these operations. In other words this language has the capability of recursive processing.

We defined above informally that the top nodes are those elements in an ERP-relation which have no immediate or indirect predecessors. It is obvious that any collection of ERP-relations has also its own top nodes. Next we define the top_nodes operation which obtains the top nodes in a given collection of ERP-relations.

Operation 2:

top_nodes: P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
top_nodes(ERP-rels) = {x|<x,y>□Σ_rel_union(ERP-rels) 3
¬ ∃ c:c□Nodes 3<c,x>□Σ_rel_union(ERP-rels)}

Because each ERP-relation is finite it means that any collection of ERP-relations is also finite. Graphically this means that a collection of ERP-relations defines a subgraph with respect to the graph corresponding to the ERP-base at hand. In this subgraph there are nodes which have no outgoing edges. These nodes are just top nodes. From the view point of the user, top nodes are often very interesting in the semantic sense.

Let us assume, for example, that the user is interested in end products made in Munich and Stuttgart. The operations top_nodes({Munich}) and top_nodes({Stuttgart}) give these end products for the user. These operations yield the node sets {p5,p6} and {p14}, respectively. If the user is interested in end products of the production process which consists of the products made in Munich or Stuttgart he can express this with the operation top_nodes({Munich,Stuttgart}). In this case the operation yields the node set {p5,p6}. It is worth noting that the product p14 is not in a top node position in the subgraph of the sample ERP-base restricted by the set of ERP-relations {Munich,Stuttgart}.

Above we defined that the bottom nodes in an ERP-relation are those elements which have no immediate and thus nor indirect successors. Analogously with the top nodes, any collection of ERP-relations has also its own bottom nodes. Next we define the bottom_nodes operation which obtains the bottom nodes in a given collection of ERP-relations.

Operation 3:
bottom_nodes: P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
bottom_nodes(ERP-rels) = {y|<x,y>□ Σ_rel_union(ERP-rels) 3
¬ ∃ c:c□Nodes 3<y,c>□ Σ_rel_union(ERP-rels)}

We can interpret bottom nodes in the graphical visualization as follows. Bottom nodes in the subgraph defined by a collection of ERP-relations have no incoming edges. Let us assume, for example, that the user is interested in starting products only from the view point of Stuttgart. He can get these starting products with the operation bottom_nodes({Stuttgart}) which yields the node set {p1,p8,p15}. If the user is interested in starting products of Stuttgart or Berlin, he can get these products by the operation bottom_nodes({Stuttgart, Berlin}) which obtains the node set {p1,p15,p16,p17,p18,p19}.

The content of an ERP-relation or a collection of ERP-relations is often very interesting from the view point of the user. This is due to the fact that the grouping principle used in constructing ERP-relations has essential meaning in the semantic sense. Intuitively each

value belonging to content of an ERP-relation or a collection of ERP-relations corresponds to one node in its graphical visualization. The content of a collection of ERP-relations is defined by the function nodes as follows.

Operation 4:
nodes: P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
nodes(ERP-rels) = {x|<x,y>□ Σ_rel_union(ERP-rels)} ≈
{y|<x,y>□ Σ_rel_union(ERP-rels)}

Related to our example the operations nodes({Stuttgart}) and nodes({Stuttgart,Berlin}) give the node sets {p7,p14,p1,p8,p15} and {p7,p14,p1,p8,p15,p9,p16,p17,p18,p19}, respectively.

Predecessors and successors among elements in any collection of ERP-relations play an important role in the semantic sense. These concepts have usually a direct interpretation in the real world from the view point of the user. Therefore our goal is to offer for the user a set of predecessors and successors-based operations in terms of which the user is able to express complex queries flexibly. For example, if we have two elements x and y in our sample ERP-base so that y is a predecessor of x it means that the product x is included in the product y - maybe indirectly. In general, the predecessors of x in a given collection of sample ERP-relations contain all those products in the construction of which the product x is needed in some way. If the element y is a predecessor of the element x it means that x is a successor of the element y. Related to our example, the successors of the element y in a given collection of ERP-relations are those elements which are needed in the construction of the product y in a way or another.

The immediate environment of an element in an ERP-base is often in a special position with respect to the element at hand. In order to manipulate the immediate environment of an element we define the operations im_successors and im_predecessors, which in a collection of ERP-relations give the immediate successors or predecessors of the element, respectively.

Operation 5:
im_successors: Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
im_successors(Element,ERP-rels) =
{y|<Element,y>□ Σ_rel_union(ERP-rels)}

Related to our sample ERP-base im_successors(p14,{Munich}) = {} and im_successors(p14,{Munich,Stuttgart}) = {p7,p15}

Operation 6:
im_predecessors: Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
im_predecessors(Element,ERP-rels)=
{x|<x,Element>□ Σ_rel_union(ERP-rels)}

Related to our sample ERP-base im_predecessors(p1,{Munich}) = {p4,p3,p2} and im_predecessors(p1,{Munich,Stuttgart}) = {p4,p3,p2,p7}. The operation im_predecessors(p1, {Munich,Stuttgart}) produces the answer for the following question: 'What products, which are made in Munich or Stuttgart, need immediately the product p1'. It is obvious that the im_successors and im_predecessors operations are approriate only if the condition Element□nodes(ERP-rels) holds between their arguments.

The operations successors and predecessors are used to search all successors and predecessors of an element in a given collection of ERP-relations, respectively. This means that we have to find indirect relationships among elements, too. In turn this means that recursion is a natural way to define these operations. Next we define these operations.

Operation 7:

successors: Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
successors(Element,ERP-rels) =
A ≈ ≈$_{i\square A}$successors(i,ERP-rels)
where A= im_successors(Element,ERP-rels)
, if A ≠ □
□ , if A = □

If related to our example the user wants to get those products in Munich which are necessary immediately or indirectly for constructing the product p3 he can use the operation successor(p3,{Munich}) for this purpose. This operation yields the node set {p1,p10,p11,p14,p12}. If we want to consider the production process of the product p3 from the view point of the whole company we can do it by the operation successor(p3, {Munich,Stuttgart,Berlin}). In this case the node set {p1,p10,p11,p14,p12, p7,p15,p8,p17,p19} is obtained.

Operation 8:

predecessors: Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
predecessors(Element,ERP-rels) =
A ≈ ≈$_{i\square A}$predecessors(i,ERP-rels)
where A= im_predecessors(Element,ERP-rels)
, if A ≠ □
□ , if A = □

Let us assume that the user in the context of our sample ERP-base wants to know which products made in Berlin or Stuttgart need the starting product p17 in some way. The operation for this is predecessors(p17,{Berlin,Stuttgart}) which yields the node set {p9,p8,p7,p14}. The operation predecessors(p17,{Berlin,Stuttgart,Munich}) defines those products in the whole company which need the starting product p17 immediately or indirectly. In this case the node set {p9,p8,p7,p14,p3,p5,p6} is given.

In many cases it would be desirable to know successors or predecessors related to a set of elements (nodes) instead of one element. Let us imagine the following practical situation in our example company. One of the suppliers of the company is not able to deliver its products to the company, i.e. these products are starting products from the view point of the company. Now it would be very nice to find all those products in the production process on which this change has an immediate or indirect effect. Next we define the operations union_of_successors and union_of_predecessors to give successors and predecessors related to a node set in a given collection of ERP-relations.

Operation 9:

union_of_successors: P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
union_of_successors(Node-set,ERP-rels) =
≈$_{i\square Node\text{-}set}$successors(i,ERP-rels)

In our sample ERP-base those products which are needed immediately or indirectly in constructing the products p4 or p3 can be defined by the operation

union_of_successors({p4,p3}, {Munich,Stuttgart,Berlin}). The result of this operation is the node set {p1,p2,p10,p11,p14, p12,p13,p7,p8,p17,p19}.

Operation 10:
union_of_predecessors: P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
union_of_predecessors(Node-set,ERP-rels) =

$$\approx_{i \square \text{Node-set}} \text{predecessors}(i,\text{ERP-rels})$$

In our sample ERP-base those products which need the starting products p17 or p18 or p19 in their production processes immediately or indirectly can be defined by the operation union_of_predecessors({p17,p18,p19}, {Munich,Stuttgart,Berlin}). In this case the result {p9,p8,p7,p14,p3,p5,p6} is produced.

The user is also very often interested in the common successors or predecessors of the given node set. For these purposes we define the operations intersection_of_successors and intersection_of_predecessors.

Operation 11:
intersection_of_successors: P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
intersection_of_successors(Node-set,ERP-rels) =

$$\leftrightarrow_{i \square \text{Node-set}} \text{successors}(i,\text{ERP-rels})$$

The product p14 has been made in Stuttgart and the product p4 in Munich. If we want to know all common component products included in both products we can get these products with the operation intersection_of_successors({p14,p4},{Munich,Stuttgart,Berlin}) which gives the node set {p1,p10,p11,p12}.

Operation 12:
intersection_of_predecessors: P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
intersection_of_predecessors(Node-set,ERP-rels) =

$$\leftrightarrow_{i \square \text{Node-set}} \text{predecessors}(i,\text{ERP-rels})$$

The products p13 and p10 are starting products in Munich and the product p17 is a starting product in Berlin. If we want to find all those products of the company which contain the starting components p13, p10 and p17 we can do it by the operation intersection_of_predecessors({p13, p10,p17}, {Munich,Stuttgart,Berlin}). The operation yields the node set {p5,p6}.

The operations difference_of_predecessors and difference_of_successors are used to find those predecessors or successors of a given node set which are not predecessors or successors of another node set. As above these operations are defined in a collection of ERP-relations. Next we define these operations.

Operation 13:
difference_of_successors: P(Nodes) ∞ P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
difference_of_successors(Node-set1,Node-set2,ERP-rels) =

{x|x□union_of_successors(Node-set1,ERP-rels)3 x□union_

Let us assume that the user wants to know those products in the company the construction of which is necessary from the view point of the production of the product p5 but which do not be necessary from the view point of the production of the product p6. We can use the operation
difference_of_successors({p5},{p6},{Munich,Stuttgart,Berlin}) for this. The result of the operation is the node set {p4,p2}.

Operation 14:
difference_of_predecessors:P(Nodes) ∞ P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes)
difference_of_predecessors(Node-set1,Node-set2,ERP-rels) =
{x|x□union_of_predecessors(Node-set1,ERP-rels)3 x□union_

If the user wants to know those products in the company which need immediately or indirectly one or more from the starting products {p10,p11,p12} in Munich but none of the starting products {p16,p17,p18,p19} in Berlin he can give the operation difference_of_predecessors({p10, p11,p12},{p16,p17,p18,p19},{Munich, Stuttgart,Berlin}) for this. This operation yields the node set {p1,p2,p4}.

The common feature for all node-oriented operations, i.e. the operations operation2, ... ,operation14, is that the range sets of the signatures of these operation are P(Node). This just means that these operations yield sets consisting of elements of an ERP-base. In the graphical visualization the elements correspond to nodes.

4.3. Path-oriented operations.

It is typical of node-oriented operations that no order among nodes included in their results is not defined. Very often the user is also interested in the chain of those nodes which is needed to connect the given nodes with each other in an ERP-base. For this we define path-oriented operations. In other words the order among nodes is essential in all path-oriented operations. This means that we can use tuples in the mathematical sense to represent paths. Formally, a path from the node node1 to the node noden is presented by a tuple <node1,node2, ... ,noden>, where the immediate relationship between nodei and nodei+1 (i□{1, ... ,n-1}) exists. In the formal sense, a path in an ERP-base belongs to the set T(Nodes) and a collection of paths to the set P(T(Nodes)) (see Chapter 2).

In order to be a path between two elements or nodes in an ERP-base it means that the elements of the path have the transitive relationship with each other i.e. the one is a successor of the other. Therefore we mention explicitly what kind of relation we assume among arguments of path-oriented operations to hold. With the operation path_set(NodeA,NodeB, ERP-rels) we find all paths which are possible to construct between nodes NodeA and NodeB in a given collection of ERP-relations. We assume that the node NodeB is a successor of the NodeA, i.e. NodeB□successors(NodeA), or the node NodeA is a predecessor of the node NodeB.

Operation 15:
path_set: Nodes ∞ Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(T(Nodes))
path_set(NodeA,NodeB,ERP-rels) =
{<node1,node2, ... ,noden>| <node1,node2, ... ,noden> □ T(Nodes):node1=NodeA 3 noden=NodeB 3 ∀i□{1, ... ,n-1}:<nodei,nodei+1>□Rel 3 Rel□ERP-rels}

In spite of the above functional definition it is obvious that the construction of paths is a typical recursive operation (compare e.g. our prototype definition in [31]). Let us assume that, in the sample case, the user wants to know all paths from the product p5 to the product p1 provided the whole production process happens in Munich. For this he can use the operation path_set(p5,p1,{Munich}) which yields the tuple set {<p5,p4,p1>,<p5,p4,p2,p1>,<p5,p3,p1>}. If we have the same problem but the production process is permitted to happen both in Munich and Stuttgart we can in this case use the operation path_set(p5,p1,{Munich,Stuttgart}). Now the result {<p5,p4,p1>,<p5,p4,p2,p1>,<p5,p3,p1>,<p5,p3,p14,p7,p1>} is obtained.

Sometimes the user is interested in some particular subsets of the set constructed by path_set above. He can be interested in those paths in which the number of the nodes is minimal (i.e. the shortest paths) or maximal (i.e. the longest paths) or which consist of the certain number of nodes (i.e. the first and the last node of the paths are in a given distance from each other). For these purposes we define the operation min_length_path_set(NodeA,NodeB,ERP-rels), max_length_path_set(NodeA,NodeB,ERP-rels) and dist_length(NodeA,NodeB,ERP-rels,Dist), respectively. As in the operation 15 we assume that the node NodeB is a successor of the NodeA or the node NodeA is a predecessor of the node NodeB.

Operation 16:
min_length_path_set: Nodes ∞ Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(T(Nodes))
min_length_path_set(NodeA,NodeB,ERP-rels) =
{tuple|tuple□path_set(NodeA,NodeB,ERP-rels)3
∀tuple1(≠tuple)□path_set(NodeA,NodeB,ERP-rels):length(tuple1)≥ length(tuple)}
/*See the notational convention 4 in Chapter 2*/

Let us assume that the user wants related to our example to find the shortest paths from the product p5 to the product p1 provided the production process can happen both in Munich and Stuttgart. For this he can use the operation min_length_path_set(p5,p1,{Munich,Stuttgart}) which gives the tuple set {<p5,p4,p1>,<p5,p3,p1>}.

Operation 17:
max_length_path_set: Nodes ∞ Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(T(Nodes))
max_length_path_set(NodeA,NodeB,ERP-rels) =
{tuple|tuple□path_set(NodeA,NodeB,ERP-rels)3
∀tuple1(≠tuple)□path_set(NodeA,NodeB,ERP-rels):length(tuple1)≤length(tuple)}

Let us continue the same example for the operation 16 but now look for the longest paths. This can be done by the operation max_length_path_set(p5,p1,{Munich,Stuttgart}) which gives the tuple set {<p5,p3,p14,p7,p1>}.

Operation 18:
dist_length: Nodes ∞ Nodes ∞ P(P(Nodes ∞ Nodes)) ∞ I^+ ∅ P(T(Nodes))
dist_length(NodeA,NodeB,ERP-rels,Distance) =
{tuple|tuple□path_set(NodeA,NodeB,ERP-rels):length(tuple)=Distance}

Let us continue the same example for the operation 16 but now look for the paths which consist of four elements i.e. three edges is needed in the graphical visualization from the node

p5 to the node p1. For this task we need the operation dist_length(p5,p1, {Munich,Stuttgart},4) which gives the tuple set {<p5,p4,p2,p1>}.

So far all operations intended for processing paths have been based on paths between two nodes. From the view point of the user it would be often useful if he could get paths which exist between two node sets. Next we define the operation paths_between_nodesets(NodeSet1,NodeSet2, ERP-rels) which constructs all possible paths from the nodes in NodeSet1 to the nodes in NodeSet2 in a given collection of ERP-relations.

Operation 19:

paths_between_nodesets: P(Nodes) ∞ P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(T(Nodes))
paths_between_nodesets(NodeSet1,NodeSet2, ERP-rels) = $\approx_{ps \square S}$ ps
where S = {path_set(i,j,ERP-rels)|i□NodeSet1 3 j□NodeSet2}

It is worth noting that the set S above is a value in the set P(P(T(Nodes))) whereas the expression $\approx_{ps \square S}$ps is a value in the set P(T(Nodes)). Let us assume that the user wants to construct all possible paths from the end products p5 and p6 of the company to the products p1 and p8. This can be done by the operation paths_between_nodesets({p5,p6},{p1,p8}, {Munich,Stuttgart, Berlin}), which gives the tuple set {<p5,p4,p1>,<p5,p4,p2,p1>,<p5,p3,p1>, <p5,p3,p14,p7,p1>,<p5,p3,p14,p7,p8>,<p6,p3,p1>,<p6,p3,p14,p7,p1>,<p6,p3,p14,p7,p8>}.

All operations defined in this chapter can be characterized as ERP-based operations because they are based only on information available in the ERP-base. In fact we have two kinds of operations: those which involve transitive computation and those which do not. The former can be called transitive ERP-based operations and the latter non-transitive ERP-based operations. Conceptually, transitive ERP-based operations need recursion whereas non-transitive ERP-based operations do not. The only relation-oriented operation or the ∑_union-operation and the operations top_nodes, bottom_nodes, nodes, im_successors and im_predecessors from the node-oriented operations are non-transitive ERP-based operations. All other operations are transitive ERP-based operations. In Chapter 5 we give a functional language in which we can intermix transitive ERP-based operations and non-transitive ERP-based operations with each other.

Because several set operations are included in our language we consider different roles of these operations. The ∑_union-operation is the only operation which the user does not use explicitly. However all other operations contain implicitly this operation because it defines the scope in which computation of the operation is performed. As we have seen the argument containing ERP-relations activates the ∑_union-operation in the context of other operations. In fact the ∑_union-operation gives always some subset of immediate relationships in the ERP-base. In the formal sense it is defined by generalizing the union operation of the relational algebra, i.e. this operation is relation-oriented.

The operations from 9 to 14 are defined in terms of the operations successors, predecessors and conventional set operations. Because we allow in our functional language (Chapter 5) also conventional set operations it is obvious that these operations do not belong to the minimal set of operations of our language. These operations have been included for user convenience, because they appear as natural primitives in many query formulations (see sample queries below). It is also typical that the set operations included in these operations have to be defined among a finite number of sets. It is obvious that without these operations many query formulations would be essentially larger, and more effort to write and read them would be required. In other words, the definition of these operations is related to the classical trade-off between the minimal set of operations and the useful set of operations. Analogously, for example, the relational algebra could be defined without the join operation because it can always be expressed with a product operation followed by a selection operation. However, the join operation is convinient to use and thus it has an obvious practical value.

It is very easy to note that all operations except for ∑_union-operation return either a node set or a tuple set as their value. From the view point of the user the expressive power can be extended considerably if the traditional set operations can be applied between the results of these operations. In practice it is important that we can perform conventional set operations between the results of the operations included in our language e.g. between the results of transitive and non-transitive ERP-based operations. Our aim is to offer this possibility for the user in our recursive query language defined in Chapter 5. Therefore we assume that the usual set operations are available i.e. the operations set_union(Set1,Set2), set_difference(Set1,Set2) and set_intersection(Set1,Set2). The signatures of these operations are Set ∞ Set ∅ Set and they mean usual union, difference and intersection operations between two sets. The nature of these operations is, of course, non-recursive.

5. Functional definition of an operation-based recursive query language

Our aim is to offer for the user a recursive query language in terms of which he can make complex queries by combining the operations in Chapter 4. The most operations in Chapter 4 require recursive processing and therefore the language consisting of these operations is also recursive in nature. The alternative approach would be that the user would define recursive rules. From the view point of the user this approach would mean that he should master recursion conceptually, for example he should define how to transfer between recursion levels what is the exit rule and so on. He should describe these rules on the basis of logic and in addition in many cases he should know how these rules are processed e.g. in order to guarantee the termination of the processing. We think that it is the duty of the logic programmer to take into account these kinds of aspects - it is not the duty of the user. In our approach all recursive processing is hidden behind operations and the user can model his queries in one operation-oriented way. In Chapter 6 we will demonstrate this feature by virtue of several examples.

Next we define how operations of Chapter4 can be combined with each other. In fact we define all sequences of operations which the user can use for expressing his queries. Because we have defined exactly the semantics of the operations in Chapter 4 it means that the semantics of any legal operation sequence achieved by combining them is also exactly defined. Operation sequences constructed by combining operations in Chapter4 are called *nested operation expressions*. Next we define this concept recursively.

Definition12:
The string $f(expr_1,expr_2, ... ,expr_n)$ is a nested operation expression if f is an operation in Chapter 4 with the signature $f:D_1 \infty D_2 \infty \ ... \infty D_n \varnothing R$ and each $expr_i$ ($i \in \{1, ... ,n\}$) is either
(1) a value belonging to the set D_i or
(2) it is an expression $g(sub\text{-}expr_1, ... ,sub\text{-}expr_m)$ such that g is an operation of Chapter 4 with the signature $g: Dom_1 \infty \ ... \infty Dom_m \varnothing D_i$ and furthermore each $sub\text{-}expr_i$ $i \in \{1, ... ,m\}$ is a nested operation expression.

Our operation-based query language intended for recursive processing consists of expressions defined in definition 12. For instance, related to our example the expression union_of_successors(im_predecessors(p1, {Munich}),{Munich,Stuttgart}) is an expression generated by our language. This is due to the following facts.

• im_predecessors(p1, {Munich}) is a nested operation expression because im_predecessors is an operator in Chapter4 whose signature is Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes) and p1 Nodes and {Munich} P(P(Nodes ∞ Nodes)). On the basis (1) it is a nested expression.
• The first argument in union_of_successors(im_predecessors(p1, {Munich}),{Munich,Stuttgart}) is a nested operation expression as we have shown above and the range set of this operation is P(Nodes). On the other hand the signature of the operation union_of_successors is P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes) which means that the value yielded by im_predecessors(p1, {Munich}) is compatible with the first argument of the union_of_successors operation. In other words the condition specified by (2) holds. In addition, it is true that {Munich,Stuttgart} P(P(Nodes ∞ Nodes)).

It follows from these considerations that the expression union_of_successors(im_predecessors(p1, {Munich}),{Munich,Stuttgart}) is an expression accepted by our functional language.

The expression union_of_predecessors(path_set(p5,p1,{Munich,Stuttgart}), {Munich,Stuttgart}) is not accepted by our language although the operations union_of_predecessors and path_set have been defined in Chapter 4. This is because the signature of the operation union_of_predecessors is P(Nodes) ∞ P(P(Nodes ∞ Nodes)) ∅ P(Nodes) and the signature of the operation path_set is Nodes ∞ Nodes ∞ P(P(Nodes ∞ Nodes)) ∅ P(T(Nodes)), i.e. the value yielded by the operation path_set(p5,p1,{Munich,Stuttgart}) belongs to the set P(T(Nodes)) which is not compatible with the set P(Nodes).

6. Sample Queries

In Chapters 4 and 5 we have defined formally the expressive power related to our recursive query language. In Chapter 4 we gave several sample operations associated with our sample ERP-base. Now we consider how the user can in terms of the functional language defined in Chapter 5 express his queries in an operation-oriented way. From the view point of the user expressing queries in this language resembles expressing queries on the basis of the relational algebra. In both cases it is possible to combine different operations so that the arguments of the operations can be other operations.

Many of the operations in Chapter 4 are recursive in nature and these operations can also be implemented recursively in a straightforward way (see our prototype implementation [31]). However there are combinations of some few operations which do not contain any recursive operation. Due to our way to model information intended for recursive processing these operation combinations can also be very interesting from the view point of the user. Our sample queries 1 and 2 represent this kind of operation combinations. Our sample queries are related to the sample ERP-base above. We demonstrate that our operation-based approach offers a compact and flexible way for the user to make queries also in those cases which in the rule-based approach would require complex recursive definitions from the user. In our approach the recursive processing is invisible from the view point of the user and he can describe his queries on the basis of concepts which have natural correspondences in the universe of discourse at hand.

Sample query 1:

Let us assume that the user wants to get all products made in Munich. The user can make his query for this purpose e.g. based on the following thinking. All products in Munich can be expressed by the operation nodes({Munich}). Those products which are in Munich but which have not made in Munich can be expressed by the operation bottom_nodes({Munich}). If we take the usual set difference between these sets we get those products which have made in Munich. In other words the whole query is expressed as follows.

Expression for the sample query 1:
set_difference(nodes({Munich}),bottom_nodes({Munich}))

The result of the sample query 1 is the set {p5, p6, p4, p3, p2, p1}.

Sample query 2:
Assume next that the user wants to get those products made in Munich which are necessary for the production process of Stuttgart i.e. the products must be transferred from Munich to Stuttgart. From the view point of the user the sample query1 can be used to refer to all products made in Munich and the operation bottom_nodes({Stuttgart}) to refer to those products in Stuttgart which are not made there. By taking the set intersection between these sets the desired result is achieved.

Expression for the sample query 2:
set_intersection(
set_difference(nodes({Munich}),bottom_nodes({Munich})),
bottom_nodes({Stuttgart}))

The result of the sample query 2 is the set {p1}.

Sample query 3:
We have the following problem: Give those products of Munich which do not need the product p7 in any way and when we only consider that part of the production process which happens either in Munich or Stuttgart. All products in Munich or Stuttgart which need the product p7 in a way or another can be expressed with the operation predecessors(p7, {Munich,Stuttgart}). The operation nodes({Munich}) gives all products in Munich. In other words from the view point of the user the query for our problem can be given as follows.

Expression for the sample query 3:
set_difference(nodes({Munich}), predecessors(p7,{Stuttgart, Munich}))

The result of the sample query3 is the set {p4,p2,p13,p1,p10,p11,p12}.

Sample query 4:
Let us assume that the user is interested in those products of the company which are needed in the construction of the end products of Munich but which are not needed in the construction of the end products of Stuttgart. The end products of Munich and Stuttgart are defined by the operations top_nodes({Munich}) and top_nodes({Stuttgart}). The successors of these products express those products which are needed in their construction. The difference_of_successors operation can be used to define those successors of one node set which are not successors of another node set. Because the user is interested in all products of the company the successors must be searched among the set of all ERP-relations i.e. in the set {Munich,Stuttgart,Berlin}. This means that the functional expression 4 can be used for our original purpose.

Expression for the sample query 4:
difference_of_successors(top_nodes({Munich]),top_nodes({Stuttgart}),{Munich,Berlin, Stuttgart})

The result of the sample query 4 is the set {p4,p2,p13,p3,p14}. It is worth noting that the end product p14 of Stuttgart is one element in the result.

Sample query 5:
Next we consider only the production process which happens in Munich and Stuttgart. Now we want to get those components of the products p3 and p4 which have been made in Munich or Stuttgart. We can use the operation union_of_successors({p4,p3},{Munich,Stuttgart}) to refer to all those components which the product p4 or the product p3 can have in Munich or Stuttgart. The products which have not been made in Munich or Stuttgart can be expressed by the operation bottom_nodes({Munich,Stuttgart}). It is important to understand that bottom_nodes({Munich, Stuttgart})≠set_union(bottom_nodes({Munich}),bottom_nodes({Stuttgart})). In other words the query for our problem can be formulated by using the sample query expression 5.

Expression for the sample query 5:
set_difference(union_of_successors({p4,p3},{Munich,Stuttgart}), bottom_n

The result of the sample query 5 is the set {p2,p14,p7,p1}.

Sample query 6:
Let us assume that the user wants to get those products in Munich or Stuttgart which have been used in the construction of the product p3 and which in turn contain as components the product p17 or the product p19 (starting products of Berlin). All immediate or indirect component products of the product p3 in Munich or Stuttgart can be obtained by the operation successors(p3, {Munich,Stuttgart}). In other words the result of the operation contains in addition of the products made in Munich or Stuttgart also products transferred from Berlin where they may have been made. On the other hand the operation union_of_predecessors({p17, p19}, {Munich, Stuttgart, Berlin}) gives all those products in the company which contain the product p17 or the product p19 or both of them in some way. This means that the common elements of the results of the operations successors(p3, {Munich,Stuttgart}) and union_of_predecessors({p17, p19},{Munich, Stuttgart, Berlin}) is just what the user wants i.e. he can describe his query with the following expression.

Expression for the sample query 6:
set_intersection(successors(p3,{Munich,Stuttgart}),
union_of_predecessors({p17,p19},{Munich,Stuttgart,Berlin}))

The result of the sample expression 6 is the set {p14,p7,p8}.

Sample query 7:
In our next example the user wants to get those products made in Stuttgart which are needed in the construction of the product p6 in some way. All products which are needed immediately or indirectly in the construction of the product p6 from the view point of the whole company can be expressed by the operation successors(p6,{Munich,Stuttgart, Berlin}). Analogously with the sample query 1 the expression set_difference(nodes({Stuttgart}),bottom_nodes({Stuttgart})) gives the products made in

Stuttgart. Now we have to select only the part indicated by this expression from the result of the operation successors(p6, {Munich,Stuttgart, Berlin}), i.e. we need the following formulation for our example.

Expression for the sample query 7:
set_intersection(successors(p6,{Munich,Stuttgart,Berlin}),
set_difference(nodes({Stuttgart}),bottom_nodes({Stuttgart})))

The result of this query is the set {p14,p7}.

Sample query 8:
Let us assume that the user wants to find all possible paths which can exist between the end products of Munich and the starting products of Berlin. The results of the operations top_nodes({Munich}) and bottom_nodes({Berlin}) contain the corresponding end and starting products. The operation paths_between_nodesets can be used to construct paths between these elements. The production system has to be considered from the view point of the whole company, i.e. in the set {Berlin, Stuttgart,Munich}. This means that our example can be described with the query expression 8.

Expression for the sample query 8:
paths_between_nodesets(top_nodes({Munich}),bottom_nodes({Berlin}),
{Berlin,Stuttgart,Munich})

The result of the sample query 8 is the set {<p5,p3,p14,p7,p8,p17>,<p5,p3, p14,p7,p8,p19>,<p6,p3,p14,p7,p8,p17>,<p6,p3,p14,p7,p8,p19>}.

Sample query 9:
We construct all possible paths among the products of the company from the end products made in Munich to those products made in Munich which are necessary for the production in Berlin or Stuttgart. Analogously with the expression 2 we find those products made in Munich which are necessary for the production in Berlin or Stuttgart i.e. we need the expression set_intersection(set_difference(nodes({Munich}),bottom_nodes({Munich})), bottom_nodes({Berlin,Stuttgart})). By using the operation paths_between_nodesets to construct the paths from the nodes in the result of the operation top_nodes({Munich}) to the nodes in the result of the above expression we get the query expression 9 for our original problem.

The expression for sample query 9:
paths_between_nodesets(top_nodes({Munich}),

set_intersection(set_difference(nodes({Munich}),bottom_nodes({Munich})),bottom_nodes({
Berlin,Stuttgart})),{Munich,Stuttgart,Berlin})

The result of the sample query 9 is the set {<p5,p4,p1>,<p5,p4,p2,p1>,<p5, p3,p1>,<p5,p3,p14,p7,p1>,<p6,p3,p1>,<p6,p3,p14,p7,p1>}.

Of course we could give several further sample queries based on our functional language defined in Chapter 5 but we believe that these are enough to demonstrate how to express

queries from the view point of the user. More sample queries based on our language can be found in [37], [38].

In order to compare our operation-oriented approach with the standard rule-based approach we consider as an example how the subexpression predecessors(p7,{Stuttgart,Munich}) in sample query 3 would be specified in terms of Horn clauses. Let us image that our sample ERP-base (see Fig. 1.) is represented as facts munich(p1,p10), ... ,stuttgart(p7,p1), ... ,berlin(p9,p16), ... , berlin(p3,p19). On the basis of these facts we can define the predicate transitive_relationship(Pred_Node, Node) where Pred_Node is a predecessor node of a node Node in the scope restricted by stuttgart and munich. The predicate is defined by the rules 1-4.

(1) immediate_relationship(Pred_Node,Node) ♦ stuttgart(Pred_Node,Node).
(2) immediate_relationship(Pred_Node,Node) ♦ munich(Pred_Node,Node).
(3) transitive_relationship(Pred_Node,Node) ♦
    immediate_relationship(Pred_Node,Node).
(4) transitive_relationship(Pred_Node,Node) ♦
    immediate_relationship(Node1,Node),
    transitive_relationship(Pred_Node,Node1).

If the user has specified these rules then he can make the query transitive_relationship(X,p7) ?. The solutions of X are predecessors nodes of p7. Generally taken the above definition can produce the same solutions many times. This is due to the fact a certain node can be proven a predecessor node of a given node in several ways. If the user wants his answer as a set consisting of different predecessor nodes then the above definition must be extended considerably. Even now the number of the characters the user must enter is essentially greater than in the corresponding operational expression. It is also typical of the above definition that it has been bound to the sample scope (stuttgart, munich) and it cannot be applied in other scopes. In order to avoid these problems the above definition should be extended by similar techniques applied in the implementation of the operation predecessor (see Part II).

We think that the operation-oriented approach is more suitable to the end user than the rule-based approach. Next we consider those factors related to the rule-based approach which are cumbersome from the view point of an ordinary end user. We can use the above rule-based definition, in spite of its relative simplicity, to identify factors which the user has to master in his rule definition.

• First he must be able to formulate his rules in terms of logic. In addition he must understand the pattern matching mechanism within rules and between a query and rules. The non-professional user is not familiar with this kind of way of modelling whereas he has practiced combination of available high-level operations.

• In the context of recursive queries the user has to master recursive definition (see the above definition) conceptually well, i.e. he has to be able to formulate generally rules for transferring from one level of recursion to another and in addition he must know how recursion is terminated by some exit rule. It is obvious that in practice the construction of recursive rules, at least in complex cases, is too heavy for non-professional users. In our approach the recursive definitions have been embedded into operations i.e. the user does not need to be aware of this definition level. This means that the user in our approach utilizes recursively defined operations (see Part II) but he does not need to define these operations.

• It is typical of a rule-based approach that a specification for a query consists of many layers of definitions on different abstraction levels. Thus the user must formulate all definition levels which in the context of complex queries means large specifications. For example, the rule-based definitions corresponding to our sample queries 3-9 would be much larger than our concise expressions above. Part II, in which we give a rule-based prototype implementation for our operation-oriented language, shows also clearly how much larger specifications are needed in a rule-based approach than in our operation-oriented approach.

• Unfortunately in rule-based deductive systems it is not sufficient for the user only to master query formulation based on Horn clauses but the user has also to master completely the underlying processing mechanism. For example, the user can define the above specification in the logical sense correctly as follows. First he changes the order of the goals immediate_relationship and transitive_relationship in the body of the rule (4) and second he reorders rules so that the new rule (4) appears before the rule (3). If we in this case would use e.g. Prolog as our evaluation strategy then the computation would never terminate. Similar problems can occur, for example, also in the processing of Datalog-programs if rules create infinite relations from finite ones [5].

We think that the end user has to be able to express his recursive queries without knowing aspects related to the evaluation strategy. In our approach all operations have been implemented (see Part II) so that the user can combine them safely with each other and he does not need to know anything about the underlying evaluation strategy.

The operations of our language are on the same conceptual level as relational operations. It was recognized early in the development of query languages for relational databases that relational algebra as such is not really an end-user query language. Thus structured query languages (e.g. SQL, QUEL) and graphical interfaces (e.g. QBE) have been developed. In spite of this it is obvious that our operations are considerably more user friendly than Horn clauses as the interface. This does not, of course, preclude eventual development of even better interfaces to deductive databases. Our operations provide a sound basis and suitable primitives for e.g. graphical interfaces with an expressive power exceeding the one achieved by present approaches.

Above we have analyzed several aspects related to query formulation in rule-based deductive systems. In our opinion these aspects require skills which belong rather to a logic programmer than to a non-professional user. Therefore we apply in Part II the rule-based approach to implement our recursive operation-oriented query language defined here.

7. The Role of the Operation-oriented Recursive Query Language in Deductive Databases

It is important to note that the language defined in this paper produces only a part of intensional information in deductive databases. We can call this intensional information ERP-based information because all operations of our operation-oriented language manipulate only the ERP-base. Our approach to deductive databases contains two kinds of extensional information: the ERP-base and the Extensional Database (briefly EDB). The ERP-base contains the identifiers of objects, among which transitive relationships are computed, whereas the EDB contains the properties of these objects. It is typical of the EDB that it has been organized according to conventional data models (e.g. relational, hierarchical, network etc.).

The EDB has its own operations in terms of which data are manipulated. We call these operations EDB-based operations. Thus, in our approach a lot of intensional information is also produced by combining ERP-based operations with EDB-based operations. This means that the expressive power of the deductive database depends also on the expressive power related to the EDB. The language consisting of the ERP-based operations is needed for computing transitive relationships whereas other computing is performed by the EDB-based operations. It is important that the reader does not confuse ERP-based operations with EDB-based operations.

The clear separation of the ERP-base and the EDB affords the possibility of organizing them on the basis of different data models. We believe that in the integration of ERP-based operations and EDB-based operations we can use some EDB-based operation(s) tailored for this purpose. In Part II we show how expressions generated by our operation-oriented language can be integrated with relational expressions by using a restriction operation (see Section 5.2 in Part II). If the EDB is based on a data model other than the relational model then we have of course to use some operation(s) of that data model for integration. Also in this case we believe that this operation-oriented integration principle and techniques introduced in Part II are valid although the details are different. The work for integrating our operation-oriented language with other data models is in progress. In Part II we also discuss how this integration way can be used for implementing deductive databases on the basis of both the homogeneous and heterogeneous approach.

Aggregation related to computation of transitive relationships is also an important form of intensional information. Therefore advanced approaches to recursive queries offer some aggregation mechanism. For example, in the traversal recursion approach [20] two types of recursion are distinquished: enumeration recursion and aggregation recursion. Enumeration recursion does not contain any aggregator operator and all possible paths to connect two nodes are included in the result. In the aggregation recursion aggregator operators select one path from many alternative paths to connect two nodes. Also in terms of Agrawal's α-operation [19] it is possible to construct aggregation information in the context of transitive computation. Aggregation in this case is based on the so-called additional attribute Δ which contains the derivation history used in transitive computation. The attribute Δ is in fact a relation and it cannot be used outside the α-operation.

In our approach to deductive databases we need information both from the ERP-base and the EDB when constructing aggregation information associated with transitive computation. This means that also in this case we utilize both ERP-based and EDB-based operations. In our approach aggregation is associated with path-oriented expressions. Although paths in our approach and the attribute Δ in Agrawal's approach are represented in a different way they have an analogous function in both approaches.

We have only one generalized operation for aggregation. Because aggregation needs information from the EDB this generalized operation contains also some EDB-dependent aspects. In Part II (see Section 5.1) we give a rule-based prototype implementation for this operation in the case that the EDB has been organized according to the relational model. If the EDB would be based on some other data model (e.g. hierarchical or network) then the EDB-dependent aspects of the generalized aggregation operation would be defined in a different way. However the principle of aggregation remains similar.

In order to compute any aggregation we need all different nodes included in the result of a path-oriented expression. In addition, we need a function which associates values being

aggregated with these nodes. In the construction of this function we have to manipulate also EDB-dependent aspects because the values being aggregated are in the EDB. The role of this function in aggregation is essential because it brings together information from the ERP-base and the EDB. This is also a motivation for a map object (see Section 2.2. in Part II) in our prototype implementation. In the construction of this function we can use EDB-dependent operations, e.g. in Part II (Section 5.2.) we show how relational operations are used in the construction of this function when the EDB is based on the relational model.

The construction of the above function is the only task which contains EDB-dependent aspect. Next we consider the part of the definition of the generalized aggregation operation which is independent of the EDB, i.e. we assume that the above function is available. A function can be represented on the basis of a binary relation. Namely, a binary relation f on A∞B or explicitly f={<a1,b1>, ... , <an,bn>} where a1, ... ,an □ A and b1, ... ,bn □ B, is a function if for each a □ A there is exactly one b □ B such that <a,b> □ f. Symbolically we usually denote f: A ∅ B and instead of <a,b> □ f we write f(a)=b.

Here we refer to the EDB-independent part of the generalized aggregation operation by common_aggr and it is defined as follows when the following notational conventions are used.
• DOM(Aggr_attr) denotes the domain of the aggregation attribute Aggr_attr.
• Map is the available function which connects the ERP-based and EDB-based information in the way described above. Thus Map is formally a binary relation such that Nodes ∞ DOM(Aggr_attr) □ Map.
• $Expr_p$ denotes those path-oriented expressions which can be generated by our operation-oriented recursive query language.
• eval(Expr) denotes the result of the evaluation of a path-oriented expression Expr, i.e. Expr □ $Expr_p$ and eval(Expr) □ P(T(Nodes)).
• Aggr_type_set = {min, max, sum}. Of course, other ways of aggregation could be allowed in the definition but they would be handled in an analogous way.

common_aggr:
$Expr_p$∞Aggr_type_set∞P(Nodes∞DOM(Aggr_attr))∅P(T(Nodes)∞DOM(Aggr_attr))
common_aggr(Expr,Aggr_indicator,Map) =
{(path,i)|path□eval(Expr) 3 i,j(≠i) □ path: Map(i) ≥ Map(j)}
if Aggr_indicator = max
{(path,i)|path□eval(Expr) 3 i,j(≠i) □ path: Map(i) ≤ Map(j)}
if Aggr_indicator = min
{(path,Sum)|path□eval(Expr) 3 Sum =$\Sigma_{i \sqsubseteq path}$Map(i)}
if Aggr_indicator = sum

Consider the above aggregation principle in the context of the following sample case. Let us assume that the user wants to know all product chains from the product p5 to the product p1 provided that the whole production process happens in Munich. In addition, he wants to know the total durations related to these product chains, i.e. aggregation is needed. In this sample case the following common_aggr expression would be constructed
common_aggr(path_set(p5,p1,{Munich}),sum,{f(p1,duration1),f(p2,duration2),
f(p3,duration3), f(p4,duration4), f(p5,duration5)}).
The domain elements in the third argument (or the function which connects ERP-based and EDB-based information) are all different products appearing in the result of

eval(path_set(p5,p1,{Munich})). This evaluation in the context of our sample ERP-base produces the set {<p5,p4,p1>, <p5,p4,p2,p1>, <p5,p3,p1>}. The values duration1, ... , duration5 are imaginary values, which express production durations related to these products, and they have been obtained from the EDB. The result of the above common_aggr is the set {(<p5,p4,p1>,total_duration1), (<p5,p4,p2,p1>,total_duration2), (<p5,p3,p1>,total_duration3)} where total_duration1, ... , total_duration3 have been summed from the production durations of the products belonging to these paths. In Section 5.2 of Part II we give a prototype implementation for the generalized aggregation operation in the context of the relational model. This implementation contains both the EDB-dependent and EDB- independent parts.

8. Conclusions

The problems related to recursive queries have been considered mainly from the view point of efficient processing and at this moment there are many efficient algorithms for different recursive query types. In this paper we consider recursive queries from the view point of the non-professional user, i.e. how we can make the formulation of his recursive queries easier - also in complex cases. The conventional rule-based way to formulate complex recursive queries is too hard and cumbersome for ordinary users because it presupposes that the users are capable of strong recursive thinking. Often the user has also to know the underlying mechanism in terms of which rules are processed.

In this paper we propose an operation-oriented approach for users. We demonstrate on the basis of this approach how the user can in a flexible and compact way formulate his queries so that recursive processing is invisible from the view point of the user. He can formulate his recursive queries in terms of operations which have obvious counterparts in the real world. We specified the operations and a functional language precisely in this paper. For different needs of the user the language contains node-oriented and path-oriented operations. The relation-oriented operation is used in the definition of the scope in these operations.

In our approach data, to which the need of recursive processing is directed, is modelled as a set consisting of union-compatible binary relations. On the basis of this modelling principle we can express in the semantical sense essential information not provided by approaches based on only one binary relation. On the other hand this means that we had to define our operations so that they are able to manipulate transitive relationships among any finite number of binary relations. In [31] we give a prototype implementation for the language defined in this paper and consider how it can be integrated with relational databases. In addition we consider how the integration principles intoduced in this paper are realized in the case that our functional language is integrated with the extensional database based on the relational data model.

References

[1] Aho, A. and Ullman, J.D., Universality of Data Retrieval Languages, Proc. of the 6th ACM Symposium on Principles of Programming Languages, Texas, pp. 110-120 (1979).

[2] Abiteboul, S. and Kanellakis, P.C., Object Identity as a Query Language Primitive, Proc. of the 1989 ACM SIGMOD Conf., Oregon, pp. 159-173 (1989).

[3] Hull, R. and Su J., On Accessing Object-Oriented Databases: Expressive Power, Complexity and Restrictions, Proc. of the 1989 ACM SIGMOD Conf., Oregon, pp. 147-158 (1989).

[4] Kowalski, R., Logic as Computer Language, In: Logic Programming, Academic Press, New York, pp. 5-16 (1982).
[5] Ullman J.D., Principles of Database and Knowledge-base Systems, vol.1, Computer Science Press, New York (1989).
[6] Youn C., Henschen L.J. and Han J., Classification of Recursive Formulas in Deductive Databases, Proc. of the 1988 ACM SIGMOD Conf., Chigaco, pp. 320-328 (1988).
[7] Bancilhon F., Naive Evaluation of Recursively Defined Relations. In: Brodie M. and Mylopoulos J. (eds.), On Knowledge Base Management Systems - Integrating Database and AI Systems, Springer Verlag, pp. 165-178 (1986).
[8] Henschen L. and Naqvi S., On Compiling Queries in Recursive First-Order Data Bases, J. ACM 31, 47-85 (1984).
[9] Sterling L. and Shapiro E., The Art of Prolog, The MIT Press, London (1986).
[10] Bancilhon F., Meier D., Sagiv Y. and Ullman J.D., Magic Sets and Other Strange Ways to Implement Logic Programs, Proc. of the 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, pp. 1-15 (1986).
[11] Sacca D. and Zaniolo C., On Implementation of a Simple Class of Logic Queries for Databases, Proc. of the 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, pp. 16-23 (1986).
[12] Banchilon, F. and Ramakrishnan, R., Performance Evaluation of Data Intensive Logic Programs. In : Minker, J. (ed.), Deductive Databases and Logic Programming. Morgan Kaufman, California, pp. 439-517 (1988).
[13] Chang C.L., On the Evaluation of Queries Containing Derived Relations in Relational Databases In : Gallaire, H., Minker, J. and Nicolas J-M. (eds), Advances in Data Base Theory, vol.1, Plenum Press, New York, 235-260.
[14] Jagadish, H.V. and Agrawal, R., A Study of Transitive Closure as a Recursion Mechanism, Proc. of the 1987 ACM SIGMOD Conf., San Francisco, pp. 331-334 (1987).
[15] Zloof, M.M., Query-By-Example: Operations on Transitive Closure, RC 5526, IBM, Yorktown Hts, New York, 1975.
[16] Clemons E. , Design of an External Schema Facility to Define and Process Recursive Structures, ACM Trans. Database Syst. vol.6, no.2, 295-311 (1981).
[17] Chang C.L. and Walker A., A Prolog Programming Interface with SQL/DS. In : Kerschberg L. (ed.), the 1st International Workshop Expert Database Systems, Benjamin-Cummings, California, pp. 233-246.
[18] Guttman, A., New Features for Relational Database Systems to Support CAD Applications, Comp. Science Dept., Univ. of California, Berkley, Ph.D. Dissertation, June 1984.
[19] Agrawal, R., Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries, Proc. of the 3rd IEEE Data Engineering Conf., Los Angeles, pp. 580-590 (1987).
[20] Rosenthal, A., Heiler, S., Dayal, U. and Manola, F., Traversal Recursion: A Practical Approach to Supporting Recursive Applications, Proc. of the 1986 ACM SIGMOD Conf., Washington, pp. 166-176 (1986).
[21] Warshall, S., A Theorem on Boolean Matrices, J. ACM, vol. 9 no.1, 11-12 (1962).
[22] Warren, H.S., A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations, Commun. ACM, vol.18, no.4, 218-22O (1975).
[23] Ioannidis, Y.E., On the Computation of the Transitive Closure of Relational Operators. Proc. of the 12th VLDB Conf., Kyoto, pp. 403-411 (1986).
[24] Lu, H., New Strategies for Computing the Transitive Closure of a Database Relation, Proc. of the 13th VLDB Conf., Brighton, pp. 267-274, 1987.

[25] Lu, H., Mikkilineni, K. and Richardson, J.P., Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation, Proc. of the 3rd IEEE Data Engineering Conf., Los Angeles, pp. 112-119 (1987).
[26] Agrawal, R. and Jagadish, H.V., Direct Algorithms for Computing the Transitive Closure of Database Relations, Proc. of the 13th VLDB Conf., Brighton, pp. 255-266 (1987).
[27] Agrawal, R., Borgida, A. and Jagadish, H.V., Efficient Management of Transitive Relationships in Large Data and Knowledge Bases, Proc. of the 1989 ACM SIGMOD Conf., Oregon, pp. 253-262 (1989).
[28] Jarke, M., Linnemann, V. and Schmidt, J., Data Constructors: On the Integration of Rules and Relations, Proc. of the 11th VLDB Conf., California, pp. 227-258 (1985).
[29] Gruz I., Mendelson A.O. and Wood P.T., A Graphical Language Supporting Recursion, Proc. of the 1987 ACM SIGMOD Conf., New York, pp. 323-330 (1987).
[30] Larson, P-Å. and Deshpande V., A File Structure Supporting Traversal Recursion, Proc. of the 1989 ACM SIGMOD Conf., Oregon, pp. 243-252 (1989).
[31] Niemi, T. and Järvelin, K., The Prototype Implementation of the Operation-based Recursive Query Language and Its Integration with Relational Databases. (Report to appear in Dept. of Computer Science, Univ. of Tampere).
[32] Järvelin, K. and Niemi, T. Advanced Retrieval from Heterogeneous Fact Databases: Integration of Data Retrieval, Conversion, Aggregation and Deductive Techniques. University of Tampere, Department of Information Studies, Research Note RN-1991-1, April 1991.
[33] Leung, Y.Y. and Lee, D.L., Logic Approaches for Deductive Databases, IEEE Expert, Winter, 64-75 (1988).
[34] Brodie M.L., Research Issues in Database Specification, ACM SIGMOD Rec. vol. 13, no.3, 42-45 (1983).
[35] Niemi, T., A Seven-tuple Representation for Hierarchical Data Structures, Information Systems, vol.8, no. 3, 151-157.
[36] Niemi, T. and Järvelin, K., A Straightforward Formalization of the Relational Model, Information Systems, vol.10, no. 1, 65-76.
[37] Niemi, T. and Järvelin, K. Deductive Database Queries based on Transitive Relationships and Entity Types. University of Tampere, Department of Computer Science, Report A-1990-11, 1990.
[38] Niemi, T. and Järvelin, K. Advanced Query Formulation in Deductive Databases. Information Processing & Management, to appear.