Proceedings of the IASTED International Conference
APPLIED INFORMATICS
February 18-21, 2002, Innsbruck, Austria

CONTENT-DRIVEN SPECIFICATIONS FOR RECURSIVE PROJECT PLANNING APPLICATIONS

J.H. TER BEKKE and J.A. BAKKER
Faculty of Information Technology and Systems, Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
email: {j.h.terbekke, j.a.bakker}@its.tudelft.nl

ABSTRACT

This paper presents new effective solutions for critical path applications for directed acyclic graphs. We demonstrate that it is possible to solve these recursive problems using a data model without nested structures and a content-driven query language without explicit recursion, iteration, nesting or navigation. These solutions do not require the specification of unique start or finish nodes of the acyclic graph, which is important when data about arcs and nodes come from external sources, as might be the case in open Internet applications. The solutions do not require routing and graph depth specifications. This content-driven character of the solutions therefore makes the approach also suitable for end users.

KEYWORDS: critical path, reachability, recursion, query language, transitive closure, expressive power.

1. INTRODUCTION

Recursion is an important subject in computer science; it is studied in almost every branch. The terms transitive closure and reachability denote a well-known application area of recursion. Numerous examples of applications can be given where data-driven recursion plays a role. A global overview includes a wide range of research subjects ranging from databases (deductive, object oriented and distributed databases, etc.), compiler design, artificial intelligence and discrete mathematics to numerous practical applications as: bill of materials, project planning, inheritance graphs. Most proposed solutions are based on mathematical models in which directed graphs play an important role. However, they are implemented as computer package applications and not incorporated in the concepts of a declarative query language for database access by end users [7].

The subject of recursion is in the database area often limited to logical deduction only and it is generally not found in textbooks on databases, probably because relational algebra is not suitable for the formulation of recursion. During the 1980s therefore, a lot of research was conducted on nested relations. One of the goals was to find a declarative solution for recursive problems. However, in [10, 12] it is proven that nested algebra specifications do not offer practical solutions because of the exponential space required to compute the transitive closure. Some researchers think that an object-oriented database is the only way out for these applications because a procedural solution would be needed. However, even in textbooks on object-oriented databases (where the subject could be expected), these recursive applications hardly appear. The subject is always prominently present and is in fact indispensable for any textbook on the fundamentals of computing (see for example [1, 9]). Only recently certain forms of logical deduction were proposed for relational databases. SQL3 [18] contains new solutions, similar to graph-theoretical solutions, supporting logical deductive problem specifications. These extensions are derived from recursive Datalog rules [8].

In theory there are many advantages to declarative query language constructs. These advantages already hold for non-recursive problems:

- declarative queries are reliable, simple and short because they emphasize the 'what' and not the 'how' of problem solving;
- it is easy to determine the correctness of declarative queries;
- declarative specifications do not contain explicit recursion, iteration, nesting or navigation;
- declarative queries support ad-hoc querying; it is not necessary to determine the applications to be used beforehand.

As a consequence of our objective to offer a declarative query facility supporting recursion, we have to design the data model such that nodes and their possible connections (called arcs) are specified explicitly. Otherwise nodes and arcs cannot be addressed. Consequently the required data structure for graphs is less simple, although not really complex, compared to conventional solutions where an arc is defined implicitly through two nodes.

The used semantic data model and language are based on only simple structures, i.e. not ones that are nested. The syntax and semantics of the language is defined and extensively applied in [13]. More complex applications of semantic data modeling principles for Internet search engines, data distribution, I/O parallelism, meta modeling and version management can be found in [3, 4, 5, 15, 16]. Practical advantages of semantic data modeling principles can also be found in [17]. A survey of the principal semantic abstractions is given in [11]. A conversion tool for several popular relational DBMSs is introduced in [6].

The paper is organized as follows. In section 2 a summary is given of the most important modeling concepts of the Xplain data model. This summary is tailored to the critical path problem. Because we are primarily interested in solving practical problems, we introduce the critical path problem by means of a simple project plan for the construction of a shopping center. In section 3 some queries are given. They have been implemented and tested with data collections representing acyclic and cyclic graphs of various depths for which we have used the Xplain DBMS, version 5.6.

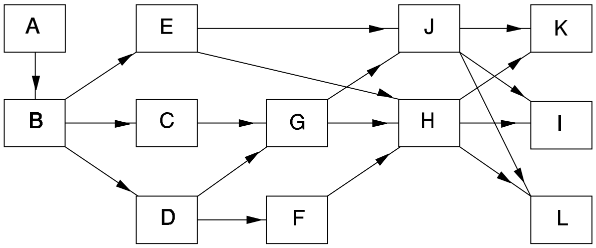

Figure 1: Project activities

2. ABSTRACTIONS

This section contains an overview of the concepts for semantic data modeling that are needed for the project planning applications. Each object will be visualized explicitly as we clearly distinguishing between identification and descriptive properties. Consequently the resulting data models gain in semantic content, while ambiguities and contradictions in the specification are avoided. Only three fundamental abstraction types with clear graphical equivalencies are required to guarantee inherent semantic integrity. These abstractions make use of the fundamental *type-attribute relationship*.

The real world is described by types (categories) of relevant objects, a type being defined as a fundamental notion. The abstraction leading to a type is called *classification*. The instances occurring in a database and triggering the recognition of a type are purely applications of the concept; the type is *not* defined by these instances. Types are represented by rectangles in diagrams. The opposite of classification is called *instantiation*.

Aggregation is defined as the collection of a certain number of types into a unit, which can be regarded as a new type. A type occurring in an aggregation is called an attribute of the new type. It is important to note the analogy with the mathematical set concept: attributes are considered as 'elements' of a type.

Aggregation allows view independence: we can discuss the obtained type (possibly as a property) without referring to the underlying attributes. By applying this principle repeatedly, a hierarchy of types can be set up. An example of a hierarchy depicting two arcs between two types is given in figure 2. Normally only composite types are represented in the visualization of the abstraction hierarchy.

If a line connects two facing rectangle sides while the aggregate type (according to its definition) is placed above its attributes, this indicates aggregation. Of course, the reverse of aggregation also exists: the description of a type as a set of certain attributes is called *decomposition*. Decomposition of a type will eventually lead to some base types. In our example database we consider the two types 'description' and 'days' as base types. A type is completely defined by a list of attributes, so we could apply the following type definitions to the activities and the prerequisite relationships shown in figure 1:

type activity = description, days.
type prerequisite = pre_activity, cur_activity.

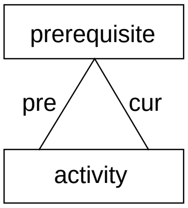

Figure 2: Aggregation hierarchy

Some instances of the project database of figure 1 are given in table 1 and table 2.

activity	description	days
A	Obtaining building licence	120
B	Access-road construction	180
C	Drilling-machine installation	3
D	Set up managerial offices	30
E	Preparation of building area	60
F	Waterworks installation	90
G	Driving piles	240
H	Building the shopping center	180
I	Electricity, sewerage	30
J	Accommodation for management	240
K	Workmanship	360
L	Parking, air conditioning	240

Table 1: Activities

prerequisite	pre_activity	cur_activity
P1	A	B
P2	B	C
P3	B	D
P4	B	E
P5	D	F
P6	C	G
P7	D	G
P8	G	H
P9	E	H
...	...	...
P17	H	K
P18	J	L
P19	H	L

Table 2: Prerequisites

Type definitions carry inherent semantics; they contain the essential properties (e.g. uniqueness of the identifications A, B, C, etc. in the activity table) and essential relationships (e.g. interdependent activities A, B, C, D, etc. mentioned under prerequisite must occur in the related activity table). Aggregation can be described using the verb *to have*. According to the above type definitions, an activity has a description and a duration (days), and a prerequisite has a current activity (cur_activity), and a previous activity (pre_activity). The delay between the start of two successive activities is subjected to the following constraint: delay ≥ pre_activity *its* days (see figure 3).

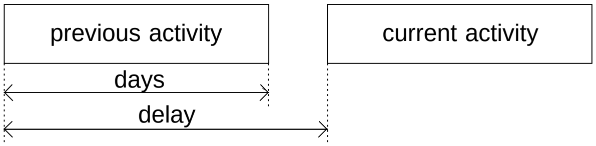

Figure 3: Duration and minimum delay

Identifications are properties denoted by type names (see table 1 above). This interpretation implies singular identifications. Attributes (not types !) may contain *roles*. Examples are cur_activity and pre_activity related to type activity. Roles are syntactically separated from the type by an underscore. Roles can be added to the aggregation connections, see figure 2.

The third kind of abstraction which is important to conceptual models is called *generalization*; here we define it as recognizing similar attributes in various types and combining them in a new type (note again the analogy with the intersection operation from mathematical set theory). We can equally discuss the new type without mentioning the underlying attributes, and the type in itself can again serve as a property in the definition of another type (i.e. it allows view independence). Generalization does not occur in our data model for project planning applications.

3. APPLICATIONS

Several queries can be presented to illustrate declarative query specifications on acyclic graphs. This section contains a few of them in order to illustrate the concepts of data manipulation in this context. Each query example starts with a title and a short description. Then the formal query specification is followed by the result on our project planning database. Finally, where necessary, an explanation is given of each line in the query.

Query 1: Critical activities

A certain minimal period is needed to complete all activities in the construction of the shopping center. This period is determined by the contents of our project planning database. The problem is split in two simple problems, as follows (see also [9]). Find for each activity its minimal preparation time (mpt). The longest path caused by all its prerequisite activities determines the starting time. Similarly we can determine for each activity its starting time before completion of all its successor activities (sbc). Critical activities have no tolerance: they have a maximal sum of minimal preparation time and latest starting time before completion. A prerequisite's minimal delay caused by the activity that precedes it is defined first.

```
extend prerequisite with delay = pre_activity its days. (1.1)
extend activity with mpt = 0.                           (1.2)
cascade activity its mpt =                              (1.3)
  max prerequisite its pre_activity its mpt + delay     (1.4)
  per cur_activity.                                     (1.5)
extend activity with sbc = days.                        (1.6)
cascade activity its sbc =                              (1.7)
  max prerequisite its cur_activity its sbc + delay     (1.8)
  per pre_activity.                                     (1.9)
value spp =                                             (1.10)
  max activity its mpt + sbc.                           (1.11)
get activity its description, mpt, sbc                  (1.12)
  where mpt + sbc = spp                                 (1.13)
  per mpt.                                              (1.14)
```

Result:

```
activity description                  mpt  sbc
    A    Obtaining building licence     0 1170
    B    Access-road construction     120 1050
    D    Set up managerial offices    300  870
    G    Driving piles                330  840
    J    Accommodation for management 570  600
    K    Workmanship                  810  360
```

Explanation:

(1.1) The minimal delay caused by the preceding activity is added to the definition of prerequisite.

(1.2) Type activity is extended with temporary attribute 'mpt' (minimal preparation time). The value for this attribute is initialized with zero for all activities (the starting point is not known beforehand so all activities may start immediately).

(1.3) The cascade update statement must be used in the case of a dependency between source and target instances. The extend statement cannot be used because of the dependency between source and target.

(1.4) The required topological ordering is clear: the value for mpt in a activity instance related to attribute pre_activity is used to calculate the maximum value of mpt in an instance related to attribute cur_activity of prerequisite. So the maximum of the related pre_activity instance (i.e. pre_activity *its* mpt) must already exist. This implies that processing must start with basic activities (their maximum is already known because of the initialization).

(1.6) Type activity is extended with a temporary attribute 'sbc' (start before completion). Each activity must start before completion of the project. The last activity is not known beforehand.

(1.7) The cascade statement is needed again, but now in the opposite direction.

(1.10) The shortest path period (spp) is determined by the maximum value for the addition.

(1.12) The result should contain the identification (which is always given) and the specified attributes, including the derived attributes.

(1.13) Only activities in the critical path are required in the result.

(1.14) Because the ordering in the result is unknown, a sorting criterion is specified. Here we sort on ascending mpt values (-mpt in the case of descending order of mpt).

It is clear that the cascade statement is indispensable for recursive applications. Its general form is:

```
cascade <subtype> its <cascade attribute> =
  <function> <maintype> its <expression>
  per <grouping attribute>.
```

The following constraints regarding this statement must be satisfied:

- <expression> must contain the <cascade attribute>, this can be determined during the automatic parsing process of the query statement. The reference of <cascade attribute> in <expression> (for example: cur_activity) must differ from the reference in the <grouping attribute> (for example: pre_activity). In the case this condition is not satisfied, the statement should be considered as a normal update statement without prescribed ordering;
- the <grouping attribute> must be identical to <subtype>, possibly with a role added;
- it is evident that all usual constraints hold, for example: types, operations and functions must be applicable and all attributes must occur in the corresponding types;
- it is only necessary to create a list of arcs such that the related <cascade attribute> value (related to a node) occurs before the <grouping attribute> value (also related to a node). This desired ordering can be determined during the query parsing process;
- the <function> must be one of the available set functions: *total* (for the sum of values), *max* (see query 1), *min* (see query 2) or the logical function *any* (see query 3).

Query 2: Minimal period between two selected activities

The previous example illustrated the *max* function in the cascade statement. Now an illustration is given of the *min* function. This example requires designation of the start and finish activities for determination of the minimal path period between these activities (here: D and L).

```
extend prerequisite with delay = pre_activity its days. (2.1)
value inf = 9999.                                       (2.2)
extend activity with est = inf.                         (2.3)
update activity "D" its est = 0.                        (2.4)
cascade activity its est =                              (2.5)
  min prerequisite its delay + pre_activity its est     (2.6)
  per cur_activity.                                     (2.7)
extend activity with sbc = inf.                         (2.8)
update activity "L" its sbc = days.                     (2.9)
cascade activity its sbc =                              (2.10)
  min prerequisite its delay + cur_activity its sbc     (2.11)
  per pre_activity.                                     (2.12)
value spp = min activity its est + sbc.                 (2.13)
get activity its description, est, sbc, spp             (2.14)
  where est + sbc = spp and est ≠ inf and sbc ≠ inf.    (2.15)
```

Result:

activity	description	est	sbc	spp
D	Setup managerial offices	0	540	540
F	Waterworks installation	30	510	540
H	Building the center	120	420	540
L	Parking, airconditioning	300	240	540

Explanation:

(2.4) Only for the starting activity D the earliest starting time (est) is initialized with zero. Other activities get the high value called inf.

(2.5) The minimal distance from the starting activity is calculated using the min function.

(2.10) The minimal distance from the finishing activity is calculated analogously

(2.14) An activity on the shortest path between the two activities has the minimal sum of the values est and mpt.

Query 3: Prerequisites of a selected activity

It can be useful to know the prerequisites of a selected activity (here: G). A prerequisite can be found via different connections in the database (the graph is generally not a simple tree structure).

```
extend activity with pre = (false).                     (3.1)
update activity "G" its pre = (true).                   (3.2)
cascade activity its pre =                              (3.3)
  any prerequisite where cur_activity its pre           (3.4)
  per pre_activity.                                     (3.5)
get activity its description, days                      (3.6)
  where pre.                                            (3.7)
```

Result:

activity	description	days
A	Obtaining building licence	120
B	Access-road construction	180
C	Drilling-machine installation	3
D	Set up managerial offices	30
G	Driving piles	240

Explanation:

(3.1) All instances will start with 'pre' extension initialized with the logical value *false*.

(3.2) Activity G (i.e. the source) is the starting point of the cascade statement.

(3.4) Attributes pre_activity and cur_activity of prerequisite determine the order of processing. Attribute

’pre’ becomes *true* if any corresponding prerequisite exists which satisfies the given condition.

(3.6) The result consists of all prerequisites incl. the source activity.

Other related queries:
In this section only a few examples of recursive applications have been given. An advantage of these content-driven solutions is that they make it possible to reuse specifications for similar queries; some examples have been presented earlier in this section to illustrate this aspect. A list of queries in this database could include also the following queries:

- Critical prerequisites of a certain activity;
- Topological ordering of activities;
- Activities following a certain activity.

CONCLUSION

Simple declarative query language solutions have been presented for the class of recursive problems on directed acyclic graphs. By using two explicit semantic objects (node and arc) instead of one explicit (node) and one implicit (pair of nodes) we were able to specify easily adaptable, reusable solutions for these problems without explicit recursion, iteration, nesting, or navigation. These declarative solutions do not require specification of unique start or finish nodes of the acyclic graph, which is important when arcs and nodes come from external sources, as might be the case in open Internet applications.

REFERENCES

[1] V.S. Alagar, *Fundamentals of computing: theory and practice* (Englewood Cliffs NJ, Prentice-Hall Int., 1989).

[2] D.A. Bailey, *Java structures: data structures in Java for the principled programmer* (Boston, MacGraw-Hill, 1999).

[3] Bert Bakker and Johan ter Bekke, Foolproof query access to search engines, *Proc. 3rd Int. Conf. on Information Integration and Web-based Applications & Services (IIWAS 2001)*, Linz, Austria, 2001, 389-394.

[4] J.A. Bakker, An extended meta model for conditional fragmentation, Proc. 9th Int. Conf. on Database and Expert Systems Applications DEXA’98, Vienna, 1998, *Lecture Notes in Computer Science, 1460*, 702-715.

[5] J.A. Bakker, Semantic partitioning as a basis for parallel I/O in database management systems, *Parallel Computing, 26*, Elsevier, 2000, 1491-1513.

[6] Berend de Boer and J.H. ter Bekke, Applying semantic database principles in a relational environment, *Proc. IASTED Int. Symp. Applied Informatics* (AI2001), Innsbruck, 2001, 400-405 (see also: http://www.pobox.com/~berend/xplain2sql/index.html).

[7] T. Conally, C. Begg, A. Strachan, *Database systems: a practical approach to design, implementation and management* (Reading Mass., Addison-Wesley, 2001).

[8] G. Gardarin and P. Valduriez, *Relational databases and knowledge bases* (Reading Mass., Addison-Wesley, 1989).

[9] J.C. Molluzzo, *A first course in discrete mathematics* (Belmont CA, Wadsforth, 1986).

[10] J. Paredaens, D. Van Gucht, Converting nested algebra expressions into flat algebra expressions, *ACM Transactions on Database Systems, 17*(1), 1992, 65-93.

[11] F. Rolland, *The essence of databases* (Hemel Hempstead, Prentice Hall, 1998).

[12] D. Suciu, J. Paredaens, Any algorithm in the complex object algebra with powerset needs exponential space to compute transitive closure, *Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1994, 201.

[13] J.H. ter Bekke, *Semantic data modeling* (Hemel Hempstead, Prentice Hall, 1992).

[14] J.H. ter Bekke, Complex values in databases, *Proc. Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering*, Hong Kong, 1994, 449-455.

[15] J.H. ter Bekke, Meta modeling for end user computing, *Proc. Workshop on Data and Expert Systems Applications DEXA 1995*, London, 1995, 267-273.

[16] J.H. ter Bekke, Semantic modeling of successive events applied to version management, *Proc. Int. Symp. on Cooperative Database Systems for Advanced Applications (CODAS ’96)*, Kyoto, 1996, 32-39; also in: *Cooperative databases and Applications* (Singapore, World Scientific, 1997), 440-447.

[17] J.H. ter Bekke, Advantages of a compact semantic meta model, *Proc. 2nd IEEE Metadata Conference*, Silver Spring, 1997, http://www.computer.org/conferen/proceed/meta97/papers/jterbekke/jterbekke.html.

[18] J.D. Ullman and J. Widom, *A first course in database systems* (Upper Sadle River NJ, Prentice Hall, 1997).