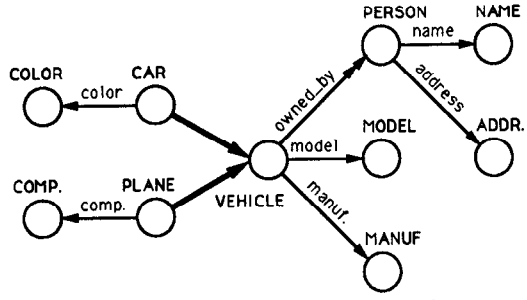
A GRAPH-ORIENTED OBJECT DATABASE MODEL

Marc Gyssens, University of Limburg (LUC), 3610 Diepenbeek, Belgium
Jan Paredaens, University of Antwerp (UIA), 2610 Antwerpen, Belgium
Dirk Van Gucht, Indiana University, Bloomington, IN 47405-4101, USA

A simple, graph-oriented database model, supporting object-identity, is presented. For this model, a transformation language based on elementary graph operations is defined. This transformation language is suitable for both querying and updates. It is shown that the transformation language supports both set-operations (except for the powerset operator) and recursive functions.

1. Introduction

Graphs have been an integral part of the database design process ever since the introduction of semantic and, more recently, object-oriented data models [6,10,14,15]. Typically, the structure (or scheme) of a database is represented as a graph. For example, the figure below displays the scheme of a vehicle-database in the Functional Data Model (FDM) [16].



Nodes in this graph represent entity classes, single (double)-arrowed edges correspond to mono (multi)-valued functions between classes, and double-sided edges indicate subclass relationships. So, whereas the scheme of the vehicle-database is specified as a graph, the instance is not, but is rather specified as a set of entity classes and functions between these classes.

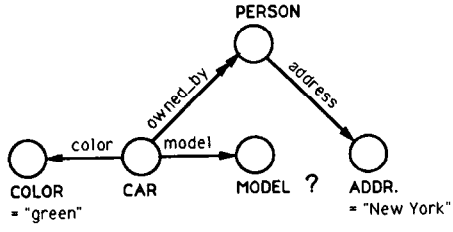
Another important feature of a data model is its associated data language [19]. It is safe to say that many semantic and object-oriented data models do not fully specify or offer this feature. This is not surprising, since most of these models were primarily introduced to aid in the database design process. To deal with the language component, typically schemes in semantic and object-oriented data models are transformed into a conceptual data model such as the relational model [19]. The required database language features then become those of the conceptual model. Certain semantic and object-oriented data models, however, are equipped with their own data language [7,10].

DAPLEX [16], for example, is a data language for FDM. The query "*List the models of green cars owned by New Yorkers*", in DAPLEX is:

```
FOR EACH CAR
SUCH THAT
      COLOR(CAR) = "Green" AND
      EXIST PERSON IN IS_OWNED_BY(CAR)
            ADDRESS(PERSON) = "New York"
LIST MODEL(CAR)
```

So, unlike the database scheme, queries in DAPLEX are formulated textually and can become cumbersome. To summarize, usually very different mathematical and linguistic tools are used to describe the various components of semantic and object-oriented data models. It is our intent to show the possibility of using a single mathematical tool, namely graphs, for this purpose.

Besides the desire for a uniform data model, we were influenced by the current trend in computing which offers users powerful workstations with graphical interfaces. We believe that this technology is already playing and will continue to play a dominant role in the development environment and user interfaces of databases [17]. In fact, many researchers have already developed such graphical database tools and techniques [8,10]. For example, the figure above can be constructed through a design tool that allows the drawing of labeled nodes and arrows. A browsing facility could offer the user the facility to retrieve the information and objects related to a particular car without the need for the user to be made aware of the overall structure of the database or of irrelevant information. As a concrete example of a graphical query one can imagine the above mentioned DAPLEX query to be graphically represented as:

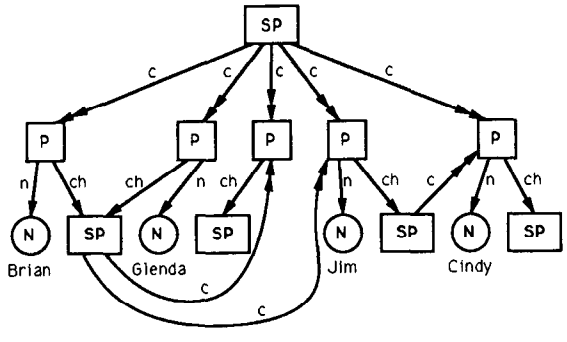


Unfortunately, the descriptions of these tools are usually rather ad hoc and infrequently based on a sound mathematical foundation. It is our intent to demonstrate that graph theory is a tool to describe a uniform data model, which at the same time offers the possibility to take advantage of graphical interfaces. In this paper, we will mainly deal with the specification of the data model, called the *Graph-Oriented Object Data model (GOOD)*.

In Section 2, we present object base schemes and instances. In Section 3, we describe the graphical transformation language of *GOOD*. In Section 4, we show that this transformation language is capable of expressing set operators (except for the powerset operator) and recursive functions. Finally in Section 5, we compare our model with some other data models and give some directions for future research.

2. The Graph-Oriented Object Database Model

It is our purpose to develop a simple and uniform model, which supports object-identity, but which is not overloaded with too many different notions. At the instance level, the data will be represented as a *directed labeled graph*. The *nodes* of this graph represent the *objects* of the database. We only distinguish between *non-printable nodes* (represented as squares) and *printable nodes* (represented as circles). In particular, we make no distinction between "atomic" objects, composed objects and set objects. As for the edges, we make no distinction between set containment, composition, generalization, specialization, etc. We only distinguish between *functional edges* (shown as "$\rightarrow$") and *non-functional edges* (shown as "$\twoheadrightarrow$"). For example, the following graph represents an object base instance for persons with their names (n) and their children (ch):



Of course, such an object base must obey certain structural constraints. These are contained in the object base scheme which can be represented by a set of productions. In this example, there are two such productions which are graphically represented as follows:

More generally, we assume there are infinitely enumerable sets of *nodes*, *non-printable object labels*, *printable object labels*, *functional object labels* and *non-functional object labels*. We also assume there is a function π which associates to each printable node a set of *constants*. A constant can be a character, a string, ..., but also a drawing, a graph, a table, etc. We now define:

Definition 2.1

An object base scheme is a five-tuple $\mathcal{S} = (NPOL, POL, FEL, NFEL, \mathcal{P})$ *with NPOL a finite set of non-printable object labels, POL a finite set of printable object labels, FEL a finite set of functional edge labels, NFEL a finite set of non-functional edge labels and* $\mathcal{P}$ *a set of productions* (L, f) *with* $L \in NPOL$ *and* $f: FEL \cup NFEL \rightarrow NPOL \cup POL$ *a partial mapping from edge labels to object labels.* ∎

Before we can proceed to the definition of an object base instance, we need some terminology. A node which is labeled by a non-printable object label is called a *non-printable node*; a node which is labeled by a printable object label is called a *printable node*. If $\mathbf{n}$ is a node, then $\lambda(\mathbf{n})$ is its object label. In addition, printable nodes may be (but do not have to be) labeled by an appropriate constant (according to the function π). A *labeled edge* is a triple $(\mathbf{m}, \alpha, \mathbf{n})$ in which $\mathbf{m}$ is a non-printable node, α a functional or non-functional edge label and $\mathbf{n}$ an arbitrary labeled node. An edge labeled with a functional edge label is called a *functional edge*; an edge labeled with a non-functional edge label is called a *non-functional edge*. We can now define:

Definition 2.2

Let $\mathcal{S} = (NPOL, POL, FEL, NFEL, \mathcal{P})$ *be an object base scheme. An object base instance over* $\mathcal{S}$ *is a pair* $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ *with:*

- $\mathbf{N}$ *a set of labeled nodes,* $\lambda(\mathbf{N})^1 \subseteq NPOL \cup POL$;
- $\mathbf{E}$ *a set of labeled edges,* $\mathbf{E} \subseteq \mathbf{N} \times (FEL \cup NFEL) \times \mathbf{N}$;
- *if* $\alpha \in FEL$ *and* $(\mathbf{m}, \alpha, \mathbf{n}_1), (\mathbf{m}, \alpha, \mathbf{n}_2) \in \mathbf{E}$, *then* $\mathbf{n}_1 = \mathbf{n}_2$;
- *for each non-printable node* $\mathbf{m} \in \mathbf{N}$ *there exists a production* $(\lambda(\mathbf{m}), f) \in \mathcal{P}$ *such that: for each edge* $(\mathbf{m}, \alpha, \mathbf{n}) \in \mathbf{E}$, $f(\alpha) = \lambda(\mathbf{n})$. ∎

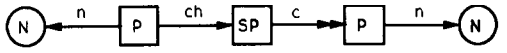
Note that some edges in a production need not correspond to edges in the object base instance. This is to allow for incomplete or unexisting information. E.g. in our example above, we used this feature of *GOOD* to express our knowledge that *Brian* and *Glenda* have two children only one of which we know by name. Note furthermore that the set $\mathcal{P}$ in the object base scheme can contain more than one production for the same non-printable object label. In this way, it is possible to model specialization and generalization.

3. A transformation language for GOOD

In this section, we present the transformation language of *GOOD*. The language contains five basic op-

[1] Functions are extended to sets in the canonical way.

erations, four of which are elementary manipulations on graphs: addition of nodes, addition of edges, deletion of edges and deletion of nodes. The specification of all these operations relies on the notion of *pattern*. A pattern is a graph used to describe subgraphs in an object base instance. Syntactically, there is no difference between a finite object base instance and a pattern. Consider e.g. the pattern:



Given the object base instance in Section 2, this pattern describes all pairs of named persons, one of which is the child of the other. More formally we have:

Definition 3.1

Let $\mathcal{S}$ be an object base scheme and let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance over $\mathcal{S}$.

- *A pattern $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ over $\mathcal{S}$ is a finite object base instance over $\mathcal{S}$.*
- *Let $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ be a pattern over $\mathcal{S}$. An embedding of $\mathcal{J}$ in $\mathcal{I}$ is a total mapping $i: \mathbf{M} \rightarrow \mathbf{N}$ preserving all labels (node labels, edge labels as well as constants).* ∎

For the pattern above, there are three embeddings into the object base instance in Section 2. They correspond to the parent-child pairs (*Brian, Jim*), (*Glenda, Jim*) and (*Jim, Cindy*), respectively.

Before we can formally define the basic operations of the transformation language, we need the notions of subscheme and subinstance.

We say that an object base scheme $\mathcal{S} = (NPOL, POL, FEL, NFEL, \mathcal{P})$ is a *subscheme* of a scheme $\mathcal{S}' = (NPOL', POL', FEL', NFEL', \mathcal{P}')$ if $NPOL \subseteq NPOL'$, $POL \subseteq POL'$, $FEL \subseteq FEL'$, $NFEL \subseteq NFEL'$ and $\mathcal{P} \subseteq \mathcal{P}'$. Similarly, an object base instance $\mathcal{I}$ over $\mathcal{S}$ is a *subinstance* of an object base instance $\mathcal{I}'$ over $\mathcal{S}'$ if $\mathcal{S}$ is a *subscheme* of $\mathcal{S}'$ and $\mathcal{I}$ is a subinstance of $\mathcal{I}'$.

In the sequel, the terms "minimal" and "maximal" will always refer to these notions of subscheme and subinstance. We are now ready to define the transformation language of *GOOD*. We start with the first operation:

Definition 3.2 (node addition)

Let $\mathcal{S}$ be an object base scheme. Let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance and $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ a pattern over $\mathcal{S}$. Let K be an arbitrary non-printable object label for which $K \notin \lambda(\mathbf{M})$. Let $\mathbf{m}_1, \ldots, \mathbf{m}_n \in \mathbf{M}$ be labeled nodes and let $\alpha_1, \ldots, \alpha_n$ be functional edge labels. The node addition

$$\mathrm{NA}[\mathcal{J}, \mathcal{S}, \mathcal{I}, K, \{(\alpha_1, \mathbf{m}_1), \ldots, (\alpha_n, \mathbf{m}_n)\}] = (\mathcal{J}', \mathcal{S}', \mathcal{I}')$$

results in a new pattern $\mathcal{J}'$, a new scheme $\mathcal{S}'$, and a new instance $\mathcal{I}'$ defined as follows:

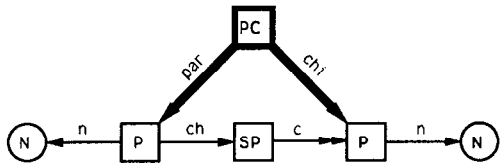
- *Let $\mathbf{m}$ be a an arbitrary printable node with $\lambda(\mathbf{m}) = K$. Then $\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ with $\mathbf{M}' = \mathbf{M} \cup \{\mathbf{m}\}$ and $\mathbf{F}' = \mathbf{F} \cup \{(\mathbf{m}, \alpha_1, \mathbf{m}_1), \ldots, (\mathbf{m}, \alpha_n, \mathbf{m}_n)\}$;*
- *$\mathcal{S}'$ is the minimal scheme of which $\mathcal{S}$ is a subscheme and over which $\mathcal{J}'$ is a pattern;*
- *$\mathcal{I}'$ is the minimal instance over $\mathcal{S}'$ for which:*
  1. *$\mathcal{I}$ is a subinstance of $\mathcal{I}'$,*
  2. *for each embedding i of $\mathcal{J}$ in $\mathcal{I}$ there exists a node $\mathbf{n}$ in $\mathcal{I}'$ such that*

     $$(\mathbf{n}, \alpha_1, i(\mathbf{m}_1)), \ldots, (\mathbf{n}, \alpha_n, i(\mathbf{m}_n))$$

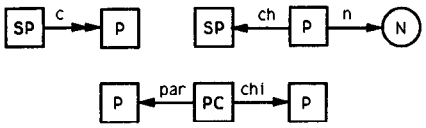
     are labeled edges of $\mathcal{I}'$,
  3. *an edge leaving a node of $\mathcal{I}$ is an edge of $\mathcal{I}$.* ∎

The last condition makes sure that the node addition is properly defined if $K \in \lambda(\mathbf{N})$ and makes sense if interpreted as an update. Observe that the node addition is always well defined if $\alpha_1, \ldots, \alpha_n$ are all different.

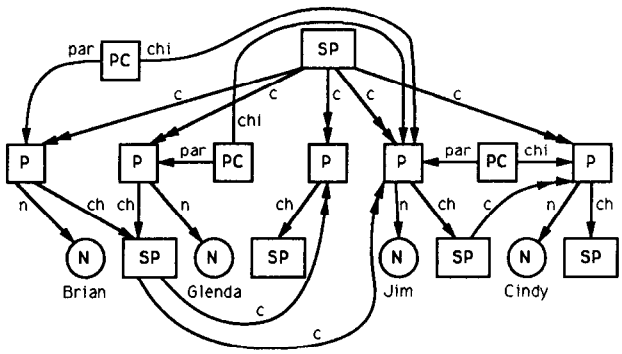
A node addition will be represented in a way well underlining the graphical nature of the model. We simply draw the pattern $\mathcal{J}'$ and mark in bold the nodes and edges not in $\mathcal{J}$. Suppose for example we want to effectively create nodes representing the named parent-child pairs occurring in the object base instance of Section 2. Using the pattern described above, this node-addition is represented as:



The resulting object base has for scheme:



and for instance:



It is readily seen that node addition can be used for both querying and for creating new objects. In this paper, however, we will not elaborate on the creation of objects for update purposes.

To the node addition also corresponds a node deletion, which in the object base instance removes nodes in all subgraphs described by a pattern:

Definition 3.3 (node deletion)

Let $\mathcal{S}$ be an object base scheme. Let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance and $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ a pattern over $\mathcal{S}$. Let $\mathbf{m}$ be a non-printable node in $\mathbf{M}$. The node deletion

$$\mathrm{ND}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathbf{m}] = (\mathcal{J}', \mathcal{S}', \mathcal{I}')$$

results in a new pattern $\mathcal{J}'$, a new scheme $\mathcal{S}'$, and a new instance $\mathcal{I}'$ defined as follows:

- *$\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ with $\mathbf{M}' = \mathbf{M} - \{\mathbf{m}\}$ and $\mathbf{F}'$ is the set of all edges in $\mathbf{F}$ not involving $\mathbf{m}$;*
- *$\mathcal{S}' = \mathcal{S}$;*
- *$\mathcal{I}'$ is the maximal subinstance of $\mathcal{I}$ such that for each embedding i of $\mathcal{J}$ in $\mathcal{I}$, $i(\mathbf{m})$ is not in $\mathcal{I}'$.* ∎

Obviously, a node deletion is always well defined.

Observe that a node deletion does not affect the scheme. Indeed, it is in general impossible to tell in advance whether a given node deletion will result in the removal of *all* nodes with a certain label. A node deletion will be denoted by drawing the pattern $\mathcal{J}$ and marking in outline the node to be deleted.

Also note that we only allowed the addition and deletion of *non-printable* nodes. This is because we conveniently assume that all necessary printable nodes are already present in the object base. However, only minor modifications are required to allow the addition or deletion of printable nodes in case one does not wish to adopt our philosophy.

The two following operations deal with edges rather than nodes:

Definition 3.4 (edge addition)

Let $\mathcal{S}$ be an object base scheme. Let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance and $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ a pattern over $\mathcal{S}$. Let $\mathbf{m}_1, \ldots, \mathbf{m}_n, \mathbf{m}'_1, \ldots, \mathbf{m}'_n \in \mathbf{M}$ be labeled nodes and let $\alpha_1, \ldots, \alpha_n$ be arbitrary edge labels. The edge addition

$$\mathrm{EA}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \{(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_n, \alpha_n, \mathbf{m}'_n)\}] = (\mathcal{J}', \mathcal{S}', \mathcal{I}')$$

results in a new pattern $\mathcal{J}'$, a new scheme $\mathcal{S}'$, and a new instance $\mathcal{I}'$ defined as follows:

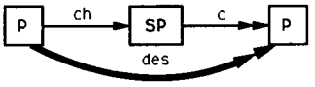
- *$\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ with $\mathbf{M}' = \mathbf{M}$ and $\mathbf{F}' = \mathbf{F} \cup \{(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_n, \alpha_n, \mathbf{m}'_n)\}$;*
- *$\mathcal{S}$ is the minimal scheme of which $\mathcal{S}$ is a subinstance and over which $\mathcal{J}'$ is a pattern;*
- *$\mathcal{I}'$ is the minimal instance over $\mathcal{S}'$ of which $\mathcal{I}$ is a subinstance, and such that for each embedding i of $\mathcal{J}$ in $\mathcal{I}'$,*

  $$(i(\mathbf{m}_1), \alpha_1, i(\mathbf{m}'_1)), \ldots, (i(\mathbf{m}_n), \alpha_n, i(\mathbf{m}'_n))$$

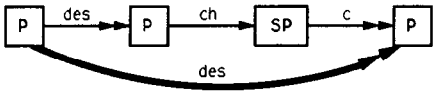
  are labeled edges in $\mathcal{I}'$. ∎

Edge addition is *not* always defined. Indeed, it is possible that $\mathcal{J}'$ contains two edges with the same label leaving the same node and arriving in nodes with different object labels. In that case, there does not exist a scheme over which $\mathcal{J}'$ is a pattern. Furthermore, $\mathcal{I}'$ does not have to be defined, even if $\mathcal{S}'$ is defined, since adding edges according to the last condition of the above definition may result in two different functional edges leaving the same node, which is not allowed by Definition 2.2.

As for node addition, we will denote an edge addition by drawing the graph $\mathcal{J}'$ and marking in bold the edges of $\mathcal{J}'$ not in $\mathcal{J}$. As an example, reconsider the person-children database of Section 2. Suppose we want to know all descendants of a person. This query can be solved by two successive edge additions. The first one is:

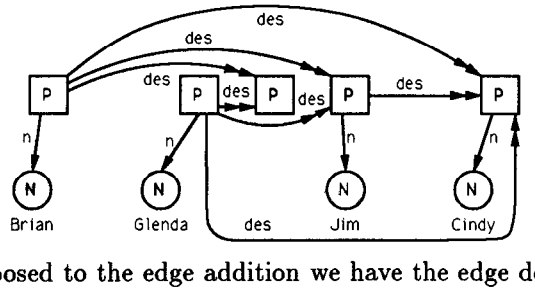


After this edge addition, *des* shows for each persons all of its children. The following edge addition then generates the transitive closure of this relation:



If we only want to retain the descendant information of persons, we can do the node deletion:

The resulting instance then becomes:



Opposed to the edge addition we have the edge deletion:

Definition 3.5 (edge deletion)

Let $\mathcal{S}$ be an object base scheme. Let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance and $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ a pattern over $\mathcal{S}$. Let $(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_n, \alpha_n, \mathbf{m}'_n)$ be labeled edges in $\mathbf{F}$. The edge deletion

$$\mathrm{ED}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \{(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_n, \alpha_n, \mathbf{m}'_n)\}] = (\mathcal{J}', \mathcal{S}', \mathcal{I}')$$

results in a new pattern $\mathcal{J}'$, a new scheme $\mathcal{S}'$, and a new instance $\mathcal{I}'$ defined as follows:

- *$\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ with $\mathbf{M}' = \mathbf{M}$ and $\mathbf{F}' = \mathbf{F} - \{(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_n, \alpha_1, \mathbf{m}'_n)\}$;*
- *$\mathcal{S}' = \mathcal{S}$;*
- *$\mathcal{I}'$ is the maximal subinstance of $\mathcal{I}$ such that for each embedding i of $\mathcal{J}$ in $\mathcal{I}$,*

  $$(i(\mathbf{m}_1), \alpha_1, i(\mathbf{m}'_1)), \ldots, (i(\mathbf{m}_n), \alpha_n, i(\mathbf{m}'_n))$$

  are not in $\mathcal{I}'$. ∎

As for node deletion, the edges to be removed are marked in outline. Also, an edge deletion is always well defined and does not affect the object base scheme.

There is still a fifth operation that remained undiscussed: the so-called *abstraction*. In *GOOD*, different nodes represent different objects, even if they cannot be distinguished by their properties actually represented in the object base. Therefore, we have introduced an abstraction that allows to define new nodes *in terms of functional or non-functional properties represented in the object base*, hence the name of the operation.

Definition 3.6 (abstraction)

Let $\mathcal{S}$ be an object base scheme. Let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance and $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ a pattern over $\mathcal{S}$. Let $\mathbf{n}$ be a non-printable node in $\mathbf{M}$. Let K be an arbitrary non-printable object label for which $K \notin \lambda(\mathbf{M})$, let $\alpha_1, \ldots, \alpha_n$ be edge labels, and let β be a non-functional edge label not occurring in $\mathcal{S}$. The abstraction

$$\mathrm{AB}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathbf{n}, K, \{\alpha_1, \ldots, \alpha_n\}]$$

results in a new pattern $\mathcal{J}'$, a new scheme $\mathcal{S}'$, and a new instance $\mathcal{I}'$ defined as follows:

- *Let $\mathbf{m}$ be a an arbitrary printable node with $\lambda(\mathbf{m}) = K$. Then $\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ with $\mathbf{M}' = \mathbf{M} \cup \{\mathbf{m}\}$ and $\mathbf{F}' = \mathbf{F} \cup \{(\mathbf{m}, \beta, \mathbf{n})\}$;*
- *$\mathcal{S}$ is the minimal scheme of which $\mathcal{S}$ is a subinstance and over which $\mathcal{J}'$ is a pattern;*
- *$\mathcal{I}'$ is the minimal instance over $\mathcal{S}'$ for which:*
  1. *$\mathcal{I}$ is a subinstance of $\mathcal{I}'$,*
  2. *for each embedding i of $\mathcal{J}$ in $\mathcal{I}$ there exists a node $\mathbf{p}$ in $\mathcal{I}'$ such that $(\mathbf{p}, \beta, i(\mathbf{n}))$ is a labeled edge of $\mathcal{I}'$,*
  3. *if $(\mathbf{p}, \beta, \mathbf{q_1})$ and $(\mathbf{p}, \beta, \mathbf{q_2})$ are both in $\mathcal{I}'$, then for each $i = 1, \ldots, n$ and for each node $\mathbf{r}$ of $\mathcal{I}$:*

$$(\mathbf{q_1}, \alpha_i, \mathbf{r}) \text{ in } \mathcal{I} \Leftrightarrow (\mathbf{q_2}, \alpha_i, \mathbf{r}) \text{ in } \mathcal{I}$$

  4. *an edge leaving a node of $\mathcal{I}$ is an edge of $\mathcal{I}$.* ∎

Intuitively, the abstraction creates K-labeled sets, each of which contains all the objects labeled $\mathbf{n}$ matching the pattern $\mathcal{J}$ and having the same $\alpha_1, \ldots, \alpha_n$-properties. The operation is always well defined.

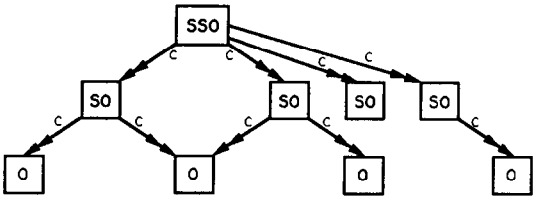
Abstraction is especially useful to reduce redundancy in the database. In the following section, we will give an example of a typical application of abstraction. As for node addition, we will denote an abstraction by drawing the graph $\mathcal{J}'$ and marking in bold the node and edge not in $\mathcal{J}$. The edge labels $\alpha_1, \ldots, \alpha_n$ will be marked as dashed arrows which do not arrive in a node.

4. The expressive power of the language

We now investigate the expressive power of the transformation language described above. First, we show that the language can simulate the (binary) set operations used in the relational algebra. (The unary operations are equally simple to express.) Then we show that, in addition, *GOOD* can compute all recursive functions on natural numbers. In Section 5, we mention some other expressiveness results.

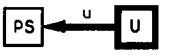
To show that *GOOD* can simulate set operations, we consider an object base with productions:

representing a set of sets of objects. A possible instance could be:

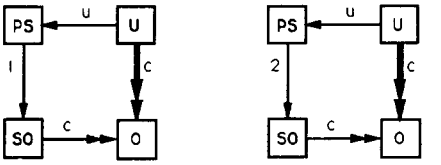


Using node addition, we can easily represent all ordered pairs of *SO*-objects:

To express union, a U-object is associated to each *PS*-object using the node addition:



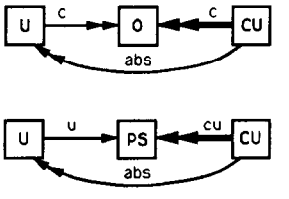
The two edge-additions:



then compute for each *PS*-object the union of the two *SO*-objects it represents. For the difference, we proceed in a similar way, except that the second edge addition is replaced by an edge *deletion*.

The above operation applied to the above instance yields 16 U-objects, because there are 16 pairs of *SO*-objects. However, interpreted as sets, only 7 different results occur. In order to "reduce" the 16 U-objects to only 7 objects, we can use the abstraction:

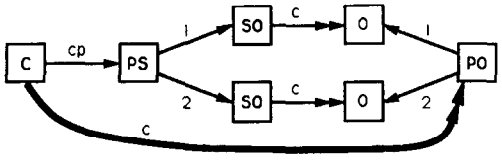
Applied to the above instance, this abstraction yields 7 *CU*-objects, each of which contains all U-objects that represent a same set. In other words, the *CU* objects represent *equivalence classes* of U-objects with respect to the equivalence relation of "representing the same set". The 7 *CU*-objects can take over the roles of the 16 U-objects if we perform the two node additions:



followed by the deletion of all U-objects.

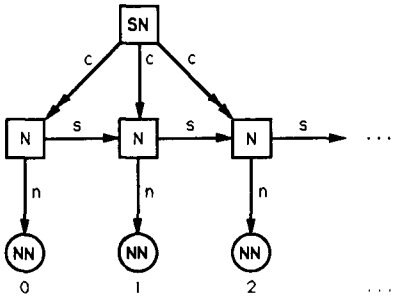
In order to construct the cartesian product, we first need to create all ordered pairs of O-objects:

As for the union, we associate to each *PS*-object a *C*-object by the appropriate node addition. The cartesian product is then computed by a simple edge addition:



We now turn to recursive functions on natural numbers. We will represent the set of natural numbers by an object base the scheme of which has productions:

and the (infinite) instance of which is:



Obviously, s represents the successor function and n associates to each number its name.

For defining recursive (partial) functions, we adopt the following definition(e.g. [21]):

Definition 4.1

Let $\mathbf{N}$ *denote the set of natural numbers.*

- *The following functions are recursive (primitives):*
  1. $\tilde{0}: \mathbf{N}^0 \to \mathbf{N}$, $\tilde{0}() = 0$,
  2. $Z: \mathbf{N} \to \mathbf{N}$, $Z(x) = 0$,
  3. $S: \mathbf{N} \to \mathbf{N}$, $S(x) = x + 1$,
  4. $\pi_i^k: \mathbf{N}^k \to \mathbf{N}$, $\pi_i^k(x_1, \ldots, x_k) = x_i$ $(1 \leq i \leq k)$.
- *Let* $f: \mathbf{N}^k \to \mathbf{N}$ *and* $g_i: \mathbf{N}^l \to \mathbf{N}$ $(1 \leq i \leq k)$ *be recursive functions. Then* $h: \mathbf{N}^l \to \mathbf{N}$ *defined by:*

  $$h(x_1, \ldots, x_l) = f(g_1(x_1, \ldots, x_l), \ldots, g_k(x_1, \ldots, x_l))$$

  is recursive. (substitution)
- *Let* $f: \mathbf{N}^k \to \mathbf{N}$ *and* $g: \mathbf{N}^{k+2} \to \mathbf{N}$ *be recursive functions. Then* $h: \mathbf{N}^{k+1} \to \mathbf{N}$ *defined by:*

  $$h(x_1, \ldots, x_k, 0) = f(x_1, \ldots, x_k)$$

  and:

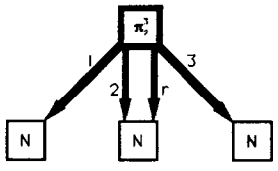
  $$h(x_1, \ldots, x_k, x_{k+1} + 1) = g(x_1, \ldots, x_{k+1}, h(x_1, \ldots, x_k, x_{k+1}))$$

  is recursive. (recursion)
- *Let* $f: \mathbf{N}^{k+1} \to \mathbf{N}$ *be total recursive. Then the function* $g: \mathbf{N}^k \to \mathbf{N}$, *defined by* $g(x_1, \ldots, x_k)$ *is the smallest number* i *for which* $f(x_1, \ldots, x_k, i) = 0$, *is recursive.* ($g(x_1, \ldots, x_k)$ *is undefined if such a number does not exist.*) *(*μ*-operator)* ∎

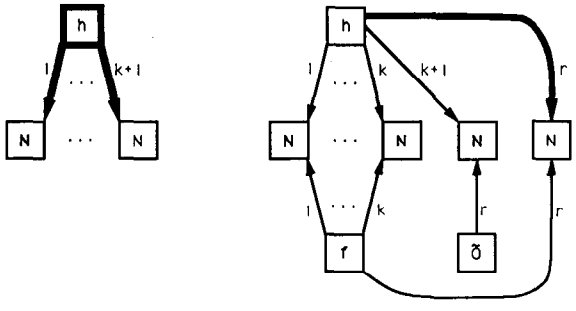
In the object base scheme for natural numbers described above, we will represent a function with k arguments as a set of ordered pairs of arity $k + 1$, the last component being the result. The arrows for the components representing the arguments of the function will be numbered consecutively; the arrow indication the result will be labeled r. We now show that, in the transformation language of *GOOD*, we can express each recursive function using the above representation of the natural numbers, and this without using the "names" of the numbers.

We start with the primitives in Definition 4.1. The $\tilde{0}$-function can be expressed by a node addition followed by a node deletion:

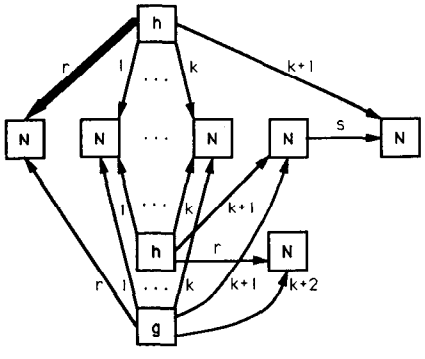
The other primitives can be constructed by one node addition. E.g. for the projection π_2^3 this is:



Since functions derived from the primitives can be partial, the r-arrows can no longer be created simultaneously with the component-arrows. Consider e.g. recursion. Then we first do a node addition, followed by an initial edge addition:



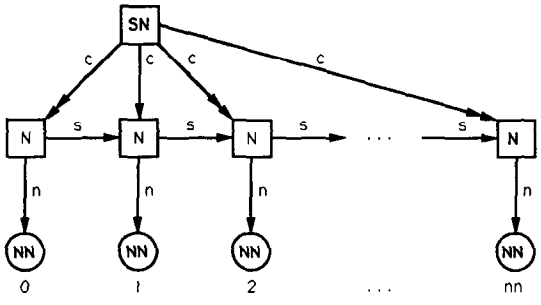
The correct result is then obtained by the recursive edge addition:



In a similar way, it can be shown that *GOOD* can handle substitution. The μ-operator, however, is a little bit more complicated to express. We give a sketch of the construction. First, we associate a new

g-object to each k-tuple of numbers, say $(x_1, \ldots, x_k)$. By a construction similar to the one above for recursion, we associate to g *all* natural numbers i for which $f(x_1, \ldots, x_k, i) = 0$. Then, we associate to each number x a new object GR which contains all numbers present in the object base instance strictly greater than x. (For this construction, we have to apply a recursive edge addition with the successor function as a basis.) Using GR, we can remove from g by edge deletion all numbers i with $f(x_1, \ldots, x_k, i) = 0$, except for the smallest.

Of course, it is not realistic to assume an infinite set of natural numbers in practical situations. In general, we will only have a finite initial segment of natural numbers, looking like:



To this finite segment, we can apply the same constructions as for the infinite segment. However, if during this construction, to find $f(n_1, \ldots, n_k)$, we need intermediate results outside this finite segment, our construction does not establish the associated value or the undefinedness of this function call. This situation can be compared to some extent with programming languages. In principle, complete programming languages can compute all recursive functions, but, in practice, the result of such a computation can be undefined, because an intermediate result may be out of range.

We may thus conclude that our transformation language is rather powerful. It offers the facility for general computation yet at the same time naturally emulates a large class of query languages. We would like to mention that although the transformation language allows for the expression of recursive queries such as found in Datalog [19], it is not powerful enough to express the powerset operator found in languages for complex objects [1,9,13].

5. Discussion and Research Directions

In this paper we introduced the *Graph-Oriented Object Database model*. We view as its strengths, its reliance on a small number of simple mathematical constructs, its object-orientation and its graphical nature. Since we mainly stressed the mathematical foundations of the model, we will now compare *GOOD* to other models and point out ongoing and future research directions.

The simplicity of the constructs used in *GOOD* is not an indicator of its "expressiveness" (see Section 4). *GOOD* can also describe designs and queries specified in other semantic and object-oriented data models. For example, it is easily seen that the vehicle database in the introduction and the corresponding DAPLEX query can be modeled in *GOOD*. In a forthcoming paper we will show how designs specified in other representative data models can be translated into corresponding *GOOD* model schemes. In addition, we could show that one can also express queries in datalog [19], stratified datalog [5,20] and datalog+functions [2,19] as well as queries specified with respect to complex objects [1,7,18] not involving the powerset operator [1,9,13].

GOOD has also some strongly object-oriented features. Object-identity is a key concept in the description of database instances. As a consequence *GOOD* allows for the natural definition of recursively specified data objects (e.g., consider the above person-children example). This feature is usually not offered in semantic models, yet is commonly available in object-oriented data models. At the data manipulation language level, *GOOD* allows the creation and destruction of objects through the node addition and deletion operations. Similar constructs were introduced in recent object-oriented query languages [3,4,11].

With respect to the use of graphs as a tool to obtain uniformity in the *GOOD* model, we want to observe that others have used different mathematical formalisms to achieve this goal. For example, recently [3] and [12] have proposed logics as their basis for the data definition and manipulation of object-oriented database models. Our choice of graphs was motivated by the observation that graphs are already heavily used in database environments and, in our opinion, will become increasingly more important as the availability of powerful workstations equipped with graphical interfaces and CASE tools increases. It is clear however that *GOOD* in its current specification has to be extended with additional graphical constructs. This is not to increase its expressiveness, but rather to make it a more user and development-oriented model. For example, it is quite easy to develop on top of *GOOD*, interfaces which give it the flavor of systems such as QBE. We believe that graph theory (because of its inherent two-dimensionality) rather than logic (which inherently one-dimensional), is a better underlying formalism to deal with such questions effectively and flexibly.

Other issues we are currently investigating are: how to express constraints (obviously in a graph-oriented fashion), how to model non-conventional applications such as CAD/CAM and hypertext, how to use GOOD as a meta data model, how to extend *GOOD* to become a complete object-oriented model (in particular the introduction of methods) and, finally, the implementation.

Acknowledgment

The authors wish to thank the program committee, and in particular Richard Hull, for drawing our attention to a flaw in the statement of the recursiveness result in an earlier version of this paper.

References

[1] S. Abiteboul, C. Beeri, "On the Power of Languages for the Manipulation of Complex Objects", *INRIA Internal Report*, 1988.

[2] S. Abiteboul, S. Grumbach, "COL: a Logic-based Language for Complex Objects", *Proc. EDBT*, 1988, pp. 271–293.

[3] S. Abiteboul, P.C. Kanellakis, "Object Identity as a Query Language Primitive", *Proc. SIGMOD Conf.*, Portland, OR, 1989, pp. 159–173.

[4] S. Abiteboul, V. Vianu, "Procedural and Declarative Database Update Language", *Proc. PODS*, Austin, TX, 1988, pp. 240–250.

[5] K. Apt, H. Blair, A. Walker, "Towards a Theory of Declarative Knowledge", *Proc. Worksh. Found. of Deductive Databases and Logic Programming*, Washington, DC, 1986, pp. 546–629.

[6] F. Bancilhon, "Object-Oriented Database Systems", *Proc. PODS*, Austin, 1988, pp. 152–162.

[7] F. Bancilhon, S. Cluet, C. Delobel, "A Query Language for the O_2 Object-Oriented Database System", *Proc. 2nd Int. Worksh. Database Programming Languages*, Gleneden Beach, OR, June 1989, pp. 93–111.

[8] K.F. Cruz, A.O. Mendelzon, P.T. Wood, "A Graphical Query Language Supporting Recursion", *Proc. SIGMOD Conf.*, San Francisco, CA, 1987, pp. 323–330.

[9] M. Gyssens, D. Van Gucht, "The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra", *Proc. SIGMOD Conf.*, Chicago, 1988, pp. 225–232.

[10] R. Hull, R. King, "Semantic Database Modeling: Survey, Applications, and Research Issues", *ACM Computing Surveys 19*, 3, September 1987, pp. 201–260.

[11] R. Hull, J.Su, "On Accessing Object-Oriented Databases: Expressive Power, Complexity, and Restrictions", *Proc. SIGMOD Conf.*, Portland, OR, 1989, pp. 147–158.

[12] M. Kifer, G. Lausen, "F-Logic, A Higher-Order Language for Reasoning About Objects, Inheritance, and Scheme", *Proc. SIGMOD Conf.*, Portland, OR, 1989, pp. 134–146.

[13] G.M. Kuper, M.Y. Vardi, "A New Approach to Database Logic", *Proc. PODS*, Waterloo, Ont., 1984, pp. 86–96.

[14] W. Kim, F.H. Lochovsky, *Object-Oriented Concepts, Databases, and Applications*, ACM Press (Frontier Series), New York, NY, 1989.

[15] J. Peckham, F. Maryanski, "Semantic Data Models", *ACM Computing Surveys 20*, 3, September 1988, pp. 153–190.

[16] D. Shipman, "The Functional Data Model and the Data Language DAPLEX", *ACM Trans. Database Syst. 6*, 1, March, pp. 140–173.

[17] M. Stonebraker, *Readings in Database Systems*, Morgan Kaufmann, S. Mateo, CA, 1988.

[18] S.J. Thomas, P.C. Fischer, "Nested Relational Structures", *The Theory of Databases*, P.C. Kanellakis, ed., JAI Press, 1986, pp. 269–307.

[19] J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. 1 and 2, Computer Science Press, Rockville, MD, 1989.

[20] A. Van Gelder, Negation as Failure Using Tight Derivations for General Logic Programs", *Proc. 3rd IEEE Symp. on Logic Programming*", 1986, pp. 127–139.

[21] K. Weihrauch, "Computability", *EATCS Monographs on Computer Science 9*, W. Brauer, G. Rozenberg, A. Salomaa (eds.), Springer-Verlag, 1987.