A Graph-Oriented Object Database Model[*]

Marc Gyssens[†] Jan Paredaens[‡] Jan Van den Bussche[‡] Dirk Van Gucht[§]

Abstract

A graph-oriented object database model (GOOD) is introduced as a theoretical basis for database systems in which manipulation as well as conceptual representation of data is transparently graph-based. In the GOOD model, the scheme as well as the instance of an object database is represented by a graph, and the data manipulation is expressed by graph transformations. These graph transformations are described using five basic operations and a method construct, all with a natural semantics. The basic operations add and delete objects and edges in function of the matchings of a pattern. The expressiveness of the model in terms of object-oriented modeling and data manipulation power is investigated.

Index terms: Database models, query languages, graph transformations, object-oriented databases, user interfaces.

[*]Preliminary versions of this paper were presented at the 9th ACM Symposium on Principles of Database Systems [16] and the 1990 ACM SIGMOD International Conference on Management of Data [17]. This work was partially supported by the DPWB of Belgium under program IT/IF/13.

[†]University of Limburg, Dept. WNI, Universitaire Campus, B-3590 Diepenbeek, Belgium. E-mail: gysm@bdiluc01.bitnet.

[‡]University of Antwerp (UIA), Dept. Math. & Comp. Science, Universiteitsplein 1, B-2610 Antwerp, Belgium. E-mail: Paredaens: pareda@wins.uia.ac.be; Van den Bussche: vdbuss@wins.uia.ac.be. Jan Van den Bussche is a Research Assistant of the NFWO.

[§]Indiana University, Comp. Science Department, Bloomington, IN 47405-4101, USA. E-mail: vgucht@cs.indiana.edu.

1 Introduction

The current database research trend is towards systems which can deal with advanced data applications that go beyond the standard "office" database application. This trend is reflected in the research on object-oriented databases [3, 20, 36].

Along with this trend, the need for better database end-user interfaces has been stressed [30, 36]. To this end, the two-dimensional nature of a computer screen should be fully exploited. It seems natural that in order to achieve these goals, graphs are used as the basic data type.

Graphs have indeed been an integral part of the database design process ever since the introduction of semantic and, more recently, object-oriented data models [19, 20, 25]. Their usage in data manipulation languages, however, is far less common. To deal with the language component, schemes in semantic and object-oriented data models are typically transformed into a conceptual data model such as the relational model. The required database language features then become those of the conceptual model.

Some semantic and object-oriented data models are equipped with their own data language. DAPLEX [29], for example, is a data language for the Functional Data Model. Even though databases in DAPLEX can be conveniently specified as graphs, queries are expressed textually and hence can become quite cumbersome.

The first graphical database end-user interfaces were developed for the relational model (e.g., QBE [38]). The earliest graphical database end-user interfaces for semantic models were associated with the Entity-Relationship Model [11, 12, 31, 35, 37]. Subsequently, graphical interfaces were developed for more complex semantic object-oriented database models [4, 14, 21, 23]. Graph-oriented end-user interfaces have also been developed for recursive data objects and queries [7, 18, 38]. Unfortunately, most interfaces using graphs as their central tool are rather limited in expressive power, as far as data manipulation is concerned.

Most object-oriented database models, on the other hand, offer computationally complete data languages. These languages are mostly non-graphical however, and therefore do not lend themselves easily as a basis for graphical user interfaces. Rather, they are expressed textually, using constructs such as path expressions, derived from object-oriented programming languages such as Smalltalk. As we will show in this paper however, the same and even greater functionality of path expressions can also be expressed graphically, using graph patterns.

It is our purpose to present an object-oriented database model in which both data representation and data manipulation are graph-based. Thereto, we introduce the *Graph-Oriented Object Database Model (GOOD)* and discuss its properties.

In GOOD, objects are represented as nodes in a graph. The edges in the graph represent the relationships and properties of the objects. This simple framework provides a convenient way to model complex object bases and take full advantage of graphical user interfaces. Object bases can be manipulated by adding or deleting nodes and edges in the graph using GOOD's graphical transformation language. The basic operations of this language are based on pattern matching; not surprisingly, they are reminiscent of graph grammars [10], and in fact even of conventional grammars (cfr. [15]). The language also includes a method mechanism, and is computationally complete. Thus, it satisfies the expressiveness conditions usually imposed on object-oriented data manipulation languages.

The GOOD model provides a theoretical basis for working with complex object bases in a transparent manner. In order to demonstrate this, an implementation of GOOD has recently been developed at the University of Antwerp [8].

This paper is further organized as follows. In Section 2 we define how graphs represent object database schemes and instances. In Section 3, we first introduce the five basic operations of our language: node addition, edge addition, node deletion, edge deletion and abstraction. We further introduce a method construct. In Section 4, we discuss additional aspects: macros, object-orientation, and computational completeness. Finally, in Section 5, we compare our approach with graph grammars, further discuss implementation matters, and mention some further work.

2 Object base schemes and instances

To introduce the concept of an object base scheme, consider the following example.

Assume that we want to specify a hyper-media system [6] storing documents which may contain text, graphics or sound information. Figure 1 shows a possible object base scheme for such a system. The graphical representation used is quite standard. The nodes in this graph represent object classes. Node labels serve as class names. User-defined classes have a rectangular shape; system-defined classes have an oval shape. We will refer to system-defined classes as *printable* classes since objects in these classes usually have a format in which they can be output to the user. For example, the rectangular-shaped node with label *Info* represents the class of nodes of information in the hyper-media system. The oval shaped node with label *String* is the class of character strings.

The single-arrowed (double-arrowed) edges in the scheme graph represent properties of and relationships between classes. They are labeled by so-called functional (multivalued) edge labels serving as names. Functional edges represent functional relationships. For example, the functional edge *created* indicates that each info node will have only one creation-date. Multivalued edges represent non-functional relationships. For example, the

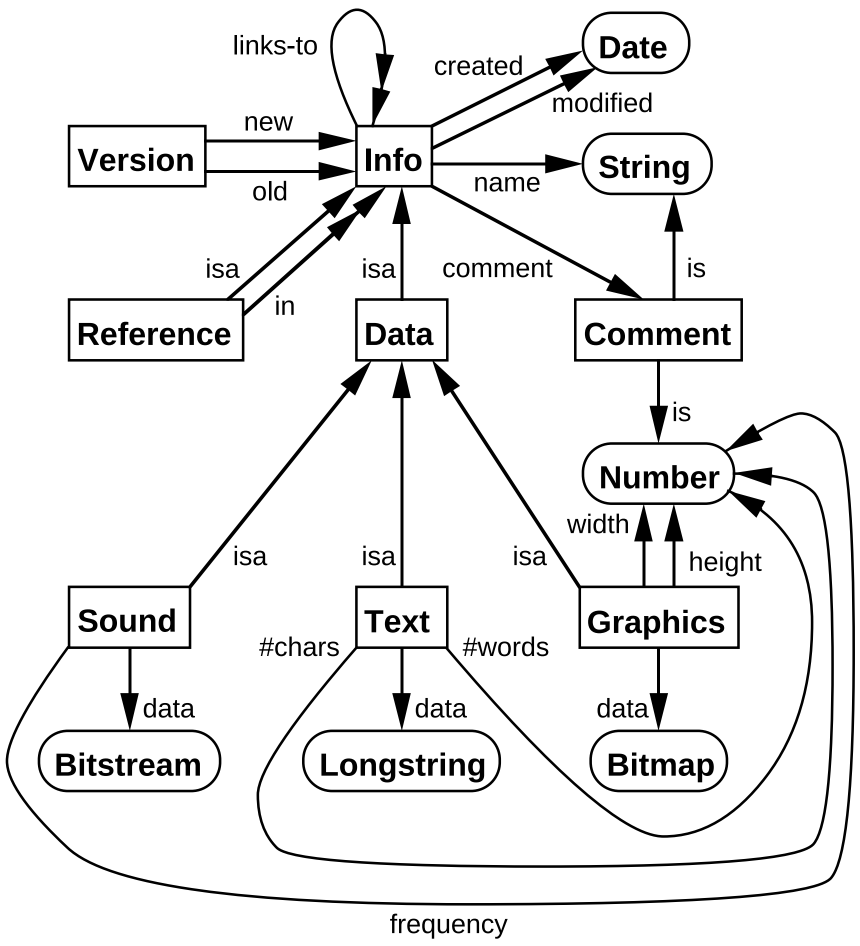

Figure 1: The hyper-media object base scheme.

edge *links-to* is multivalued; a certain info node can link to multiple other info nodes.

Let us look in more detail at the example scheme of Figure 1. An info node represents a node of information in the hyper-media system. Associated with this node are a creation date, a last modification date, a name, a comment (either a string or a number), and possibly other info nodes.

Since it is typical in hyper-media systems to have various versions of the same document, we need a way to keep track of different versions. This is facilitated with version nodes. A version node indicates that an info node has obtained a new version. The node pointed at by the edge labeled *old* indicates the old version, whereas the edge with *new* edge label points to the node corresponding to the new version.

Furthermore, we distinguish two subclasses of info nodes: *(i)* the class of data nodes containing either text, graphics, or sound data; and *(ii)* the class of reference nodes specifying references in info nodes. Since a reference may occur in multiple info nodes, the *in* label is multivalued.

Associated with a graphics node are the height and width, and the actual data stored as a bitmap. Associated with a text node are its number of words and characters and the actual data stored as a long string. Associated with a sound node are its frequency and the actual data stored as a bit stream.

We now turn to the formal definition of an object base scheme. Throughout this paper, we assume there are infinitely enumerable sets of *nodes*, *object labels*, *printable object labels*, *functional edge labels* and *multivalued edge labels*. These four sets of labels are assumed to be pairwise disjoint. Labels can be thought of as type names. In accordance with our assumption that printable classes are system-given, we also assume there is a function π associating to each printable object label the appropriate set of *constants* (e.g., characters, strings, numbers, booleans, but also drawings, graphics, sound, etc).

An object base scheme *is a five-tuple* $\mathcal{S} = (OL, POL, FEL, MEL, \mathcal{P})$ *with*

- OL *a finite set of object labels;*
- POL *a finite set of printable object labels;*
- FEL *a finite set of functional edge labels;*
- MEL *a finite set of multivalued edge labels; and*
- $\mathcal{P} \subset OL \times (MEL \cup FEL) \times (OL \cup POL)$.

An object base scheme is represented by a directed graph with two kinds of nodes: oval-shaped nodes, labeled by an element of POL, and rectangular-shaped nodes, labeled by an *element of OL, and two kinds of edges: functional edges (shown as '$\rightarrow$'), labeled by elements of FEL, and multivalued edges (shown as '$\twoheadrightarrow$'), labeled by elements of MEL. The set $\mathcal{P}$ is represented by the edges of the graph.*

We now turn to object base instances. Succinctly speaking, an object base instance defined over an object base scheme is a directed graph of objects, conforming to the structure of the scheme. The edges between the objects are individual properties of or relationships between these objects.

In Figures 2 and 3 we show an example of a hyper-media object base instance over the object base scheme shown in Figure 1.[1] First notice how each printable node has an associated constant. To make Figure 2 more readable, we have duplicated certain printable nodes. For example, the printable node with label *date* and value *Jan 12, 1990* is repeated seven times. In reality, only one such node appears in the object base instance, obviously with seven edges arriving at it. In Figure 2 we have marked the info nodes with names *Pinkfloyd* and the *The Doors* with the numbers 1 and 2 respectively. These nodes are redisplayed in Figure 3 in dotted outline. They contain the actual data nodes in these info nodes.

The info node in the left upper corner of Figure 2 represents a document about music history. It is attached with functional edges to a creation date, a last modification date, a name, and a comment node. This node is furthermore linked (via multivalued edges labeled *linked-to*) to three other info nodes representing rock history, classical music history, and jazz history, respectively.

The version node is connected to two info nodes. The *new* edge points to the info node containing the new version and the *old* edge points to the old version. Notice how the new and old info nodes are both linked to the info node containing information about the rock group *The Doors*. This reflects the property that this information is preserved across the two versions. The single reference node indicates that the info node with name *The Beatles* is a reference in the *Jazz* info node.

Some edges specified in the object base scheme can be absent from a node in the object base instance. For example, the info node with name *The Doors* has no comment associated with it. This is a convenient way to allow for incomplete or nonexisting information. There could be even info nodes without any outgoing edges. They would represent info nodes that have no known name, comment, creation date and last-modified date. Further, if these nodes have neither incoming edges, we only know their existence: no relation with other facts stored in the database is known.

[1]We should note that we do not intend to present this typically large and complex graph as such to the user. The GOOD transformation language, to be introduced in the next section, provides tractable primitives for manipulating and visualizing relevant parts of the instance graph.

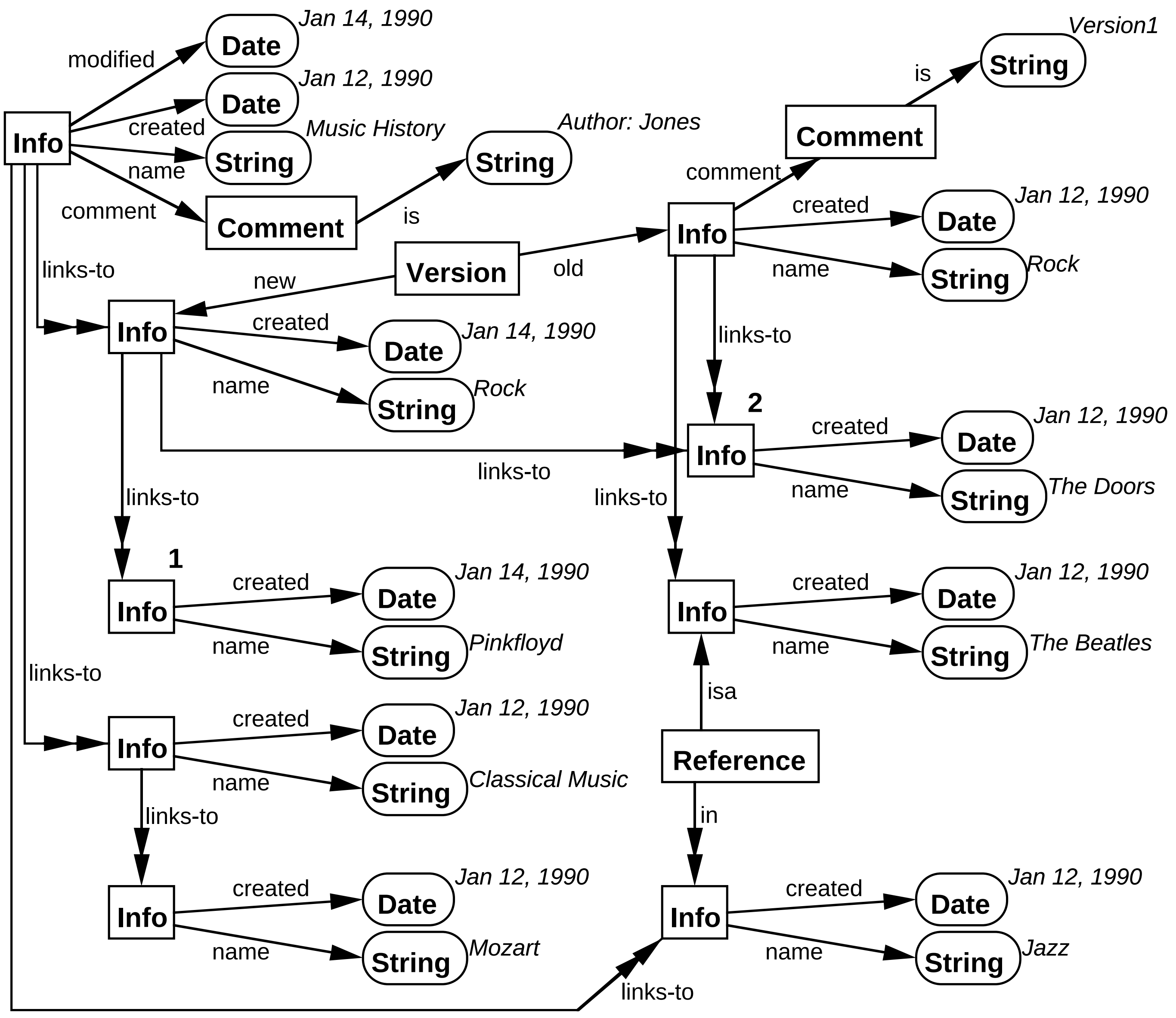

Figure 2: An example of a hyper-media object base instance.

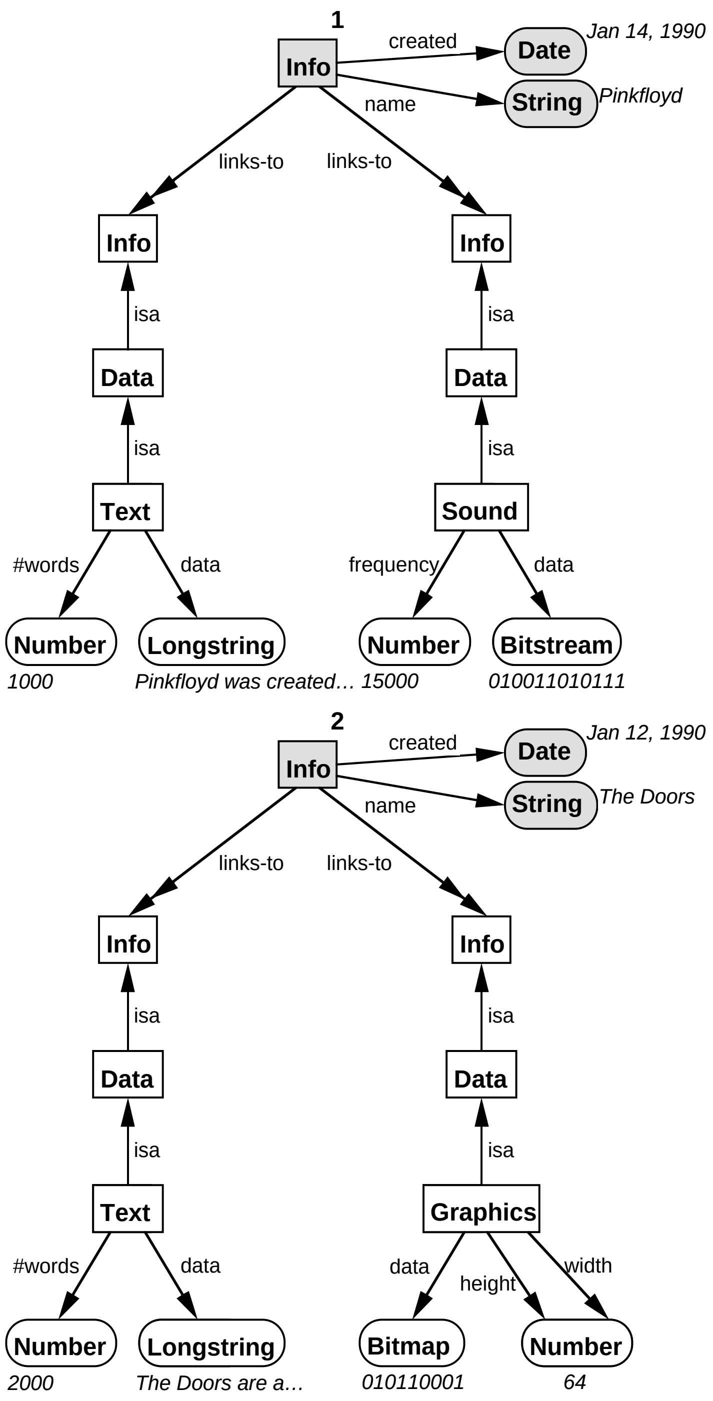

Figure 3: Continuation of the hyper-media object base instance.

As a final remark, note that the scheme and instance contain edges labeled 'isa', a name typically reserved to express inheritance in semantic data models. For simplicity, we will at this moment not formally attach special semantics to these edges. In Section 4.2 we will consider inheritance in more depth.

We are now ready to formally define object base instances.

Let $\mathcal{S} = (OL, POL, FEL, MEL, \mathcal{P})$ *be an object base scheme. Formally, an* object base instance *over* $\mathcal{S}$ *is a labeled graph* $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ *for which*

- $\mathbf{N}$ *is a* finite *set of labeled nodes; if* $\mathbf{n}$ *is a node in* $\mathbf{N}$*, then the label* $\lambda(\mathbf{n})$ *of* $\mathbf{n}$ *must be in* $OL \cup POL$*; if* $\lambda(\mathbf{n})$ *is in OL (respectively in POL), then* $\mathbf{n}$ *is also called an* object *(respectively a* printable object*);*
- *each printable node* $\mathbf{n}$ *in* $\mathbf{N}$ *can have one additional label* $\text{print}(\mathbf{n})$*, called the* print label*;* $\text{print}(\mathbf{n})$ *must be a constant in* $\pi(\lambda(\mathbf{n}))$*;*
- $\mathbf{E}$ *is a set of labeled edges; if* $\mathbf{e}$ *is a labeled edge in* $\mathbf{E}$*, then* $\mathbf{e} = (\mathbf{m}, \alpha, \mathbf{n})$ *with* $\mathbf{m}$ *and* $\mathbf{n}$ *in* $\mathbf{N}$*, the label* $\alpha = \lambda(\mathbf{e})$ *of* $\mathbf{e}$ *in* $FEL \cup MEL$*, and* $(\lambda(\mathbf{m}), \alpha, \lambda(\mathbf{n})) \in \mathcal{P}$*; if* $\lambda(\mathbf{e})$ *is in FEL (respectively in MEL), then* $\mathbf{e}$ *is called a* functional edge *(respectively a* multivalued edge*);*
- *if* $(\mathbf{m}, \alpha, \mathbf{n}_1)$ *and* $(\mathbf{m}, \alpha, \mathbf{n}_2) \in \mathbf{E}$*, then* $\lambda(\mathbf{n}_1) = \lambda(\mathbf{n}_2)$ *(i.e., the labels of all nodes connected by* α *edges to the node* $\mathbf{m}$ *have to be equal); moreover, if* $\alpha \in FEL$*, then* $\mathbf{n}_1 = \mathbf{n}_2$*.*
- *if* $\lambda(\mathbf{n}_1) = \lambda(\mathbf{n}_2)$ *is in POL and if* $\text{print}(\mathbf{n}_1) = \text{print}(\mathbf{n}_2)$ *then* $\mathbf{n}_1 = \mathbf{n}_2$*.*

Instances are graphically represented in the same way as schemes.

3 The transformation language

The GOOD data transformation language is a database language with graphical syntax and semantics. It contains five basic graph transformation operations. Four of these correspond to elementary manipulations of graphs: addition of nodes, addition of edges, deletion of nodes and deletion of edges. The fifth operation, called abstraction, is used to group objects on the basis of some of their properties. Moreover, the language includes a method call mechanism in the spirit of object-oriented database systems. All operations of the language are deterministic up to the particular choice of new objects.

Formally, the effect of a program will be defined as a transformation of the database graph, resulting in another database graph. Whether this latter database graph is only a temporary entity or actually replaces the original database graph depends on whether the

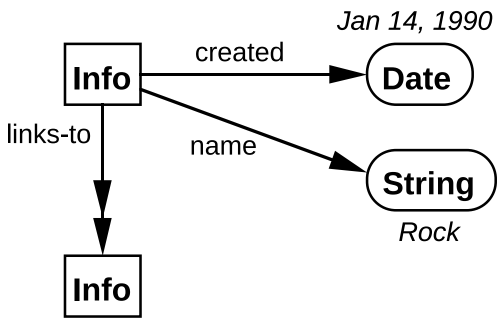

Figure 4: An example of a pattern.

transformation represents, e.g., a query or an update. The GOOD transformation language has indeed been designed in such a way that it can as well be used for querying, updating, scheme manipulations, restructuring, browsing, and visualizing parts of a complex instance. A systematic treatment of these different "modes of interpretation" is given in [2] and will not be dealt with in this paper.

Throughout the remainder of this paper, we will use the example object base instance of Figure 2 (and continued in Figure 3), unless explicitly specified otherwise.

In order to specify the operations of GOOD, we need the notion of *pattern*. A pattern is a graph used to describe subgraphs in an object base instance over a given scheme. As such, a pattern is syntactically itself an instance over that scheme.

For example, consider the graph in Figure 4. This graph is a pattern over the hypermedia object base scheme. Intuitively, it describes an info node, created on *Jan 14, 1990*, with name *Rock* which is linked to another info node.

In order to specify the subgraphs in an object base instance corresponding to a pattern, we need to introduce the concept of *matching*. The pattern in Figure 4 can be matched within the instance of Figure 2 in two different ways. A first way is shown in Figure 5; in the second way, one matches the lower Info node of the pattern with the Info node numbered '2' in Figure 2. Formally:

Let $\mathcal{S}$ be an object base scheme, let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance over $\mathcal{S}$ and let $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ be a pattern over $\mathcal{S}$. A matching *of $\mathcal{J}$ in $\mathcal{I}$ is a total mapping $i : \mathbf{M} \rightarrow \mathbf{N}$ satisfying:*

- *For each* $\mathbf{m} \in \mathbf{M}$, $\lambda(i(\mathbf{m})) = \lambda(\mathbf{m})$*;*
- *For each* $\mathbf{m} \in \mathbf{M}$, *if* $\text{print}(\mathbf{m})$ *is defined then* $\text{print}(i(\mathbf{m})) = \text{print}(\mathbf{m})$*;*
- *If* $(\mathbf{m}, \alpha, \mathbf{n}) \in \mathbf{F}$, *then* $(i(\mathbf{m}), \alpha, i(\mathbf{n})) \in \mathbf{E}$.

In words, i preserves labels and edges.

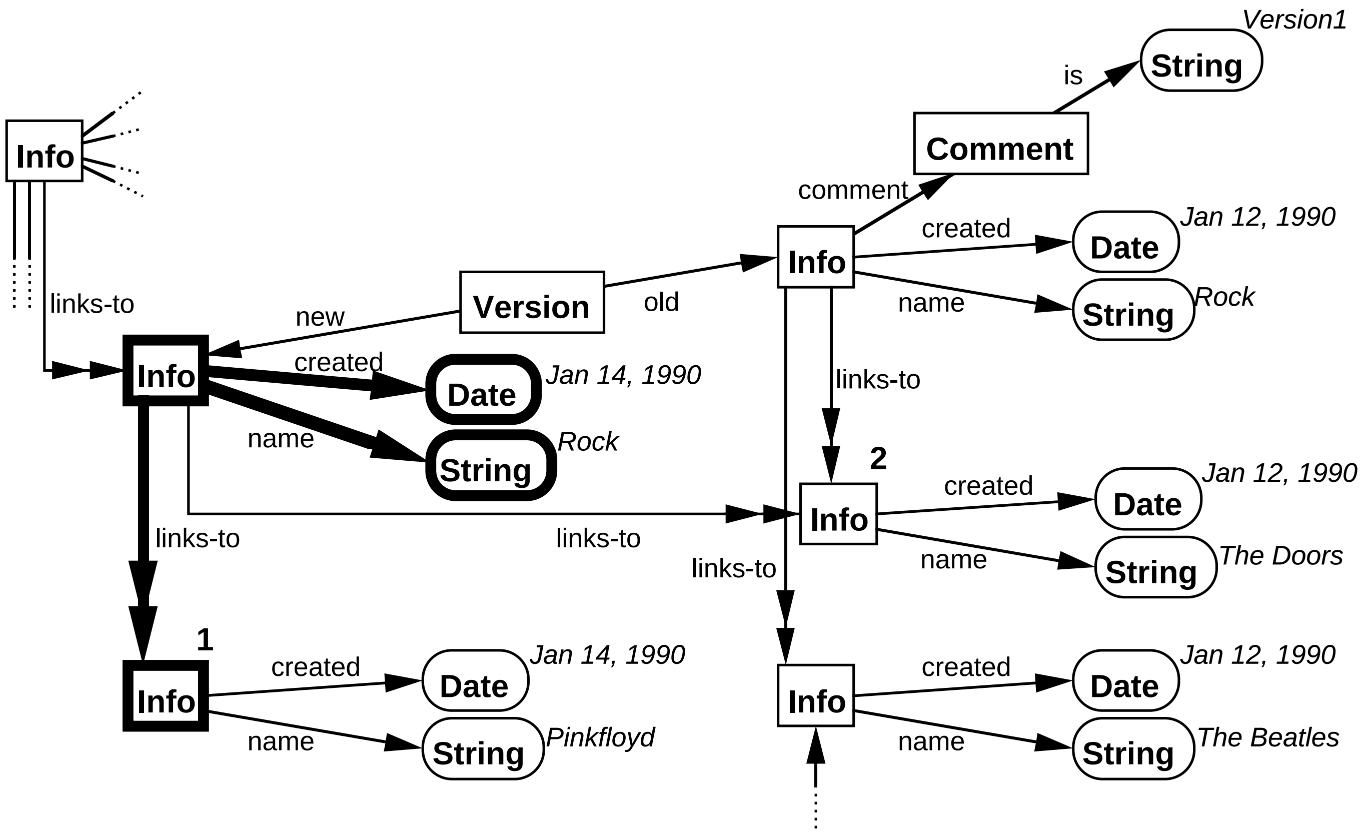

Figure 5: A first way to match the pattern of Figure 4.

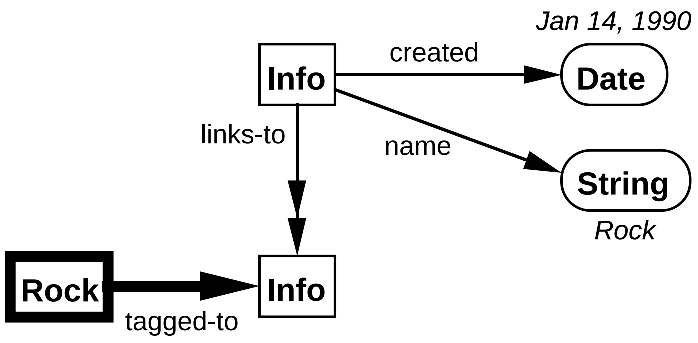

Figure 6: An example of a node addition operation.

We can now start with the discussion of the operations of the GOOD transformation language.

3.1 Node addition

Suppose that we want to locate the info nodes to which the info node with name *Rock* and date *Jan 14, 1990* is linked. Therefore, reconsider the pattern in Figure 4. As indicated before, there are two such info nodes identified by the two different matchings of this pattern in the hyper-media object base instance. The first node is the info node with name *The Doors* and created on *Jan 12, 1990*. The second node is the info node with name *Pinkfloyd* and created on *Jan 14, 1990*. A way to locate these two nodes is to associate with each one a new node. This can be accomplished with a node addition operation. The specific node addition for this example is displayed in Figure 6. This figure contains two distinguishable parts: the first part is the pattern in Figure 4 (this pattern will be called the source pattern) and the second part, indicated in bold, specifies the types of nodes and edges to be added. Intuitively, the effect of this operation is that for each matching of the source pattern, a new node and a new edge are added to the instance and linked to the proper node identified by the matching. Figure 7 shows that part of the hyper-media object base affected by this node addition.

The node addition is more general than suggested by this first example. In its general form, it can introduce objects that represent aggregates of multiple nodes in the object base instance under consideration. Consider the node addition of Figure 8. The source pattern specifies info nodes with name *Rock* and for which a creation date exists. Furthermore, these nodes have to be linked to other info nodes which also have a creation date. (As can be verified, there a four matchings of the source pattern in the hyper-media object base instance of Figure 2.) Assume that we are interested in the pairs (in general, the aggregates) of creation dates of such info nodes. The node addition under consideration derives these pairs. The four added nodes will have the node label *pair*, and will be attached

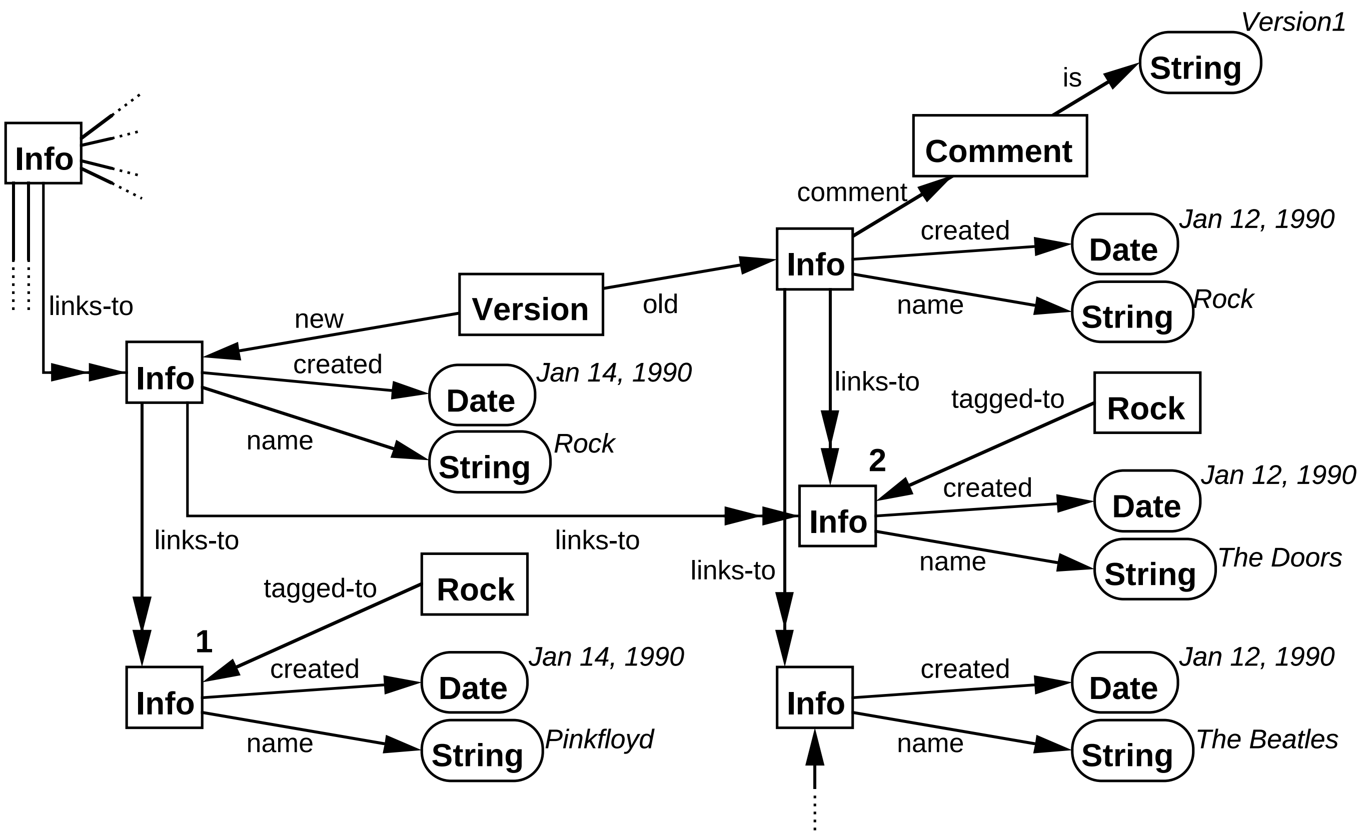

Figure 7: The result of the node addition of Figure 6.

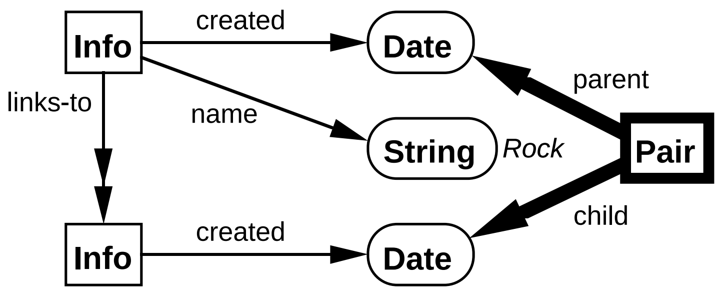

Figure 8: A node addition deriving aggregates of creation dates.

with functional edges (labeled *parent* and *child*) to the appropriate creation dates.

As can be seen from the two previous examples, node additions never introduce printable nodes. This is based on our assumption that printable nodes are system-defined and need not be explicitly added by GOOD transformation language operations. Furthermore, node additions only introduce *functional edges*. This implies that a node addition operation imposes a one to one relationship between the matchings of the source pattern, restricted to the nodes in which a bold edge arrives, and the nodes that are added by the operation.

We are now ready for the formal definition of a node addition.

Let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ *be an object base instance and* $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ *a pattern over object base scheme* $\mathcal{S}$ *(*$\mathcal{J}$ *will be called the* source pattern of the node addition*). Let* $\mathbf{m}_1, \ldots, \mathbf{m}_n$ *be nodes in* $\mathbf{M}$*. Let* K *be an object label and let* $\alpha_1, \ldots, \alpha_n$ *be different functional edge labels.*

The node addition

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

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

- $\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ *where* $\mathbf{M}'$ *is obtained by adding to* $\mathbf{M}$ *a new node* $\mathbf{m}$ *with label* K*;* $\mathbf{F}'$ *is then obtained by adding to* $\mathbf{F}$ *the labeled functional edges* $(\mathbf{m}, \alpha_1, \mathbf{m}_1), \ldots, (\mathbf{m}, \alpha_n, \mathbf{m}_n)$*;*
- $\mathcal{S}'$ *is the minimal scheme of which* $\mathcal{S}$ *is a subscheme*[2] *and over which* $\mathcal{J}'$ *is a pattern; and*
- $\mathcal{I}'$ *is the minimal object base instance (up to isomorphism) over* $\mathcal{S}'$ *for which*
  1. $\mathcal{I}$ *is a subinstance of* $\mathcal{I}'$*;*
  2. *for each matching* i *of* $\mathcal{J}$ *in* $\mathcal{I}$*, there exists a* K*-labeled 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 functional edges in* $\mathcal{I}'$*; and*

[2]Subscheme and subinstance are defined with respect to set inclusion.

```
**function** NA$[\mathcal{J}, \mathcal{S}, \mathcal{I}, K, \{(\alpha_1, \mathbf{m}_1), \ldots, (\alpha_n, \mathbf{m}_n)\}]$;
$\mathcal{I}' := \mathcal{I}$; $\mathcal{J}' := \mathcal{J}$; $\mathcal{S}' := \mathcal{S}$;
augment $\mathcal{J}'$ as in the definition;
augment $\mathcal{S}'$ with object label $K$, and for $1 \leq \ell \leq n$:
  functional edge labels $\alpha_\ell$ and triples $(K, \alpha_\ell, \lambda(\mathbf{m}_\ell))$;
for each matching $i$ of $\mathcal{J}$ in $\mathcal{I}$ do
  if not exists a $K$-labeled node $\mathbf{n}$ in $\mathcal{I}'$ with outgoing edges $(\mathbf{n}, \alpha_\ell, i(\mathbf{m}_\ell))$, $1 \leq \ell \leq n$
    then add such a node $\mathbf{n}$ and edges $(\mathbf{n}, \alpha_\ell, i(\mathbf{m}_\ell))$ to $\mathcal{I}'$;
return $(\mathcal{J}', \mathcal{S}', \mathcal{I}')$
end
```

Figure 9: Procedural semantics of node addition.

3. each edge in $\mathcal{I}'$ leaving a node of $\mathcal{I}$ is also an edge of $\mathcal{I}$.

Like the formal definitions of the other GOOD operations that will be presented in this section, the above definition of node addition is given in a "declarative" style. To show that the definition corresponds to a "procedural" semantics, we give an algorithm to compute the result of a node addition operation in Figure 9. The algorithms for the other operations are similar.

3.2 Edge addition

The node addition operation can be used to introduce new objects into an object base. The edge addition operation, in contrast, is a tool to build relationships between the objects already in an object base instance.

Consider the info node in the hypermedia object base instance with name *Pinkfloyd* and creation date *Jan 14, 1990.* This node is connected to two other info nodes (see Figure 3). These info nodes correspond to data nodes, one of type text, the other of type sound. Now assume that we want to associate the creation date of the Pinkfloyd info node with the info nodes representing the text and sound data. This can be accomplished with the edge addition operation shown in Figure 10. This figure contains two distinguishable parts: the first part is the source pattern which selects the appropriate info nodes, and the second part, indicated with the bold edge labeled *data-creation*, specifies the types of edges to be added. Figure 11 shows that part of the hyper-media object base affected by this edge addition.

Combinations of node and edge additions are useful for generating objects corresponding to sets. Assume that we want to determine the set of all info nodes created on *Jan 14, 1990*. This can be done in two steps. The first step consists of introducing an object which will denote this set. This is accomplished with the node addition shown in Figure 12 (notice

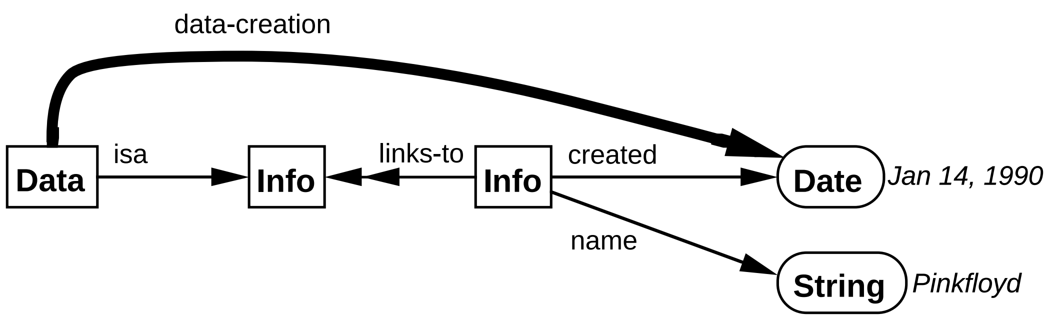

Figure 10: An example of an edge addition.

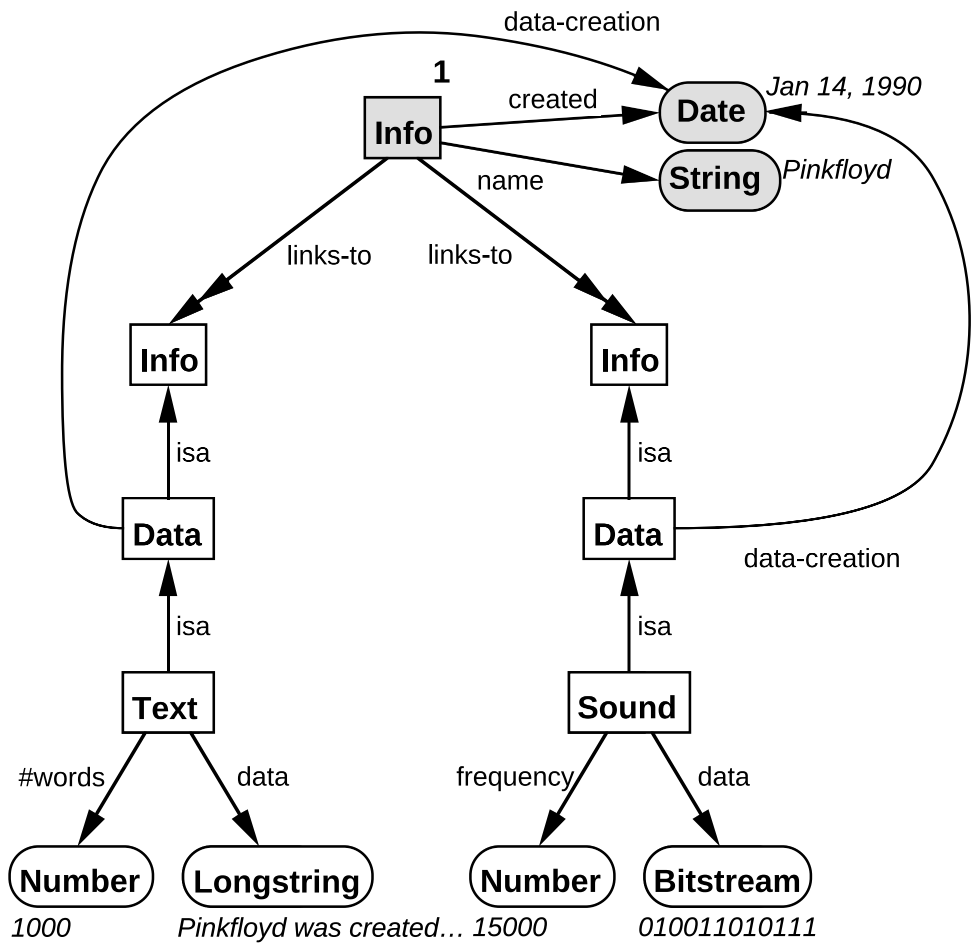

Figure 11: The result of the edge addition of Figure 10.

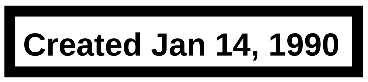

Figure 12: Adding a single node labeled *Created on Jan 14, 1990*.

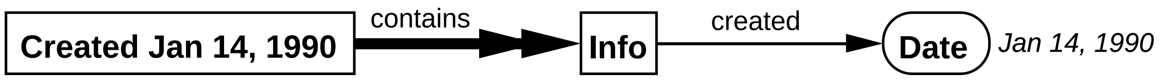

Figure 13: Linking to *Created on Jan 14, 1990* the info nodes created on Jan 14, 1990.

how the source pattern is simply the empty pattern, and consequently only one node is added here). The result of this node addition consists of the introduction of a single node with label *Created on Jan 14, 1990*. The second step consists of connecting to this newly created node all the info nodes created on *Jan 14, 1990*. This is accomplished with the multivalued edge-addition shown in Figure 13.

We are now ready for the formal definition of the edge addition.

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

The edge addition

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

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

- $\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ *where* $\mathbf{M}'$ *equals* $\mathbf{M}$ *and* $\mathbf{F}'$ *is obtained by adding to* $\mathbf{F}$ *the labeled edges* $(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_n, \alpha_n, \mathbf{m}'_n)$*;*
- $\mathcal{S}'$ *is the minimal scheme of which* $\mathcal{S}$ *is a subscheme and over which* $\mathcal{J}'$ *is a pattern;*
- $\mathcal{I}'$ *is the minimal instance over* $\mathcal{S}'$ *for which* $\mathcal{I}$ *is a subinstance of* $\mathcal{I}'$, *and such that for each matching 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* $\mathcal{I}'$.

Observe that the result of an edge addition is not defined if the addition of the required edges would yield different edges *(i)* with the same label and leaving the same node; and *(ii)* that either are functional, or arrive in nodes with different labels. Unfortunately, given an arbitrary GOOD program, i.e., a sequence of GOOD operations, statically checking the “consistency” of an edge addition in the program is undecidable in general, as can be shown using results from [1, 22]. So in general, some limited run-time checks have to be

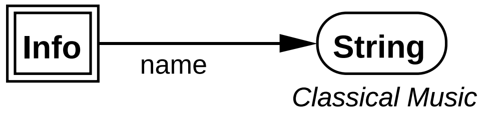

Figure 14: Example of a node deletion.

performed. In practice, one can always construct a program in such a way that the edge additions are guaranteed to succeed.

3.3 Node deletion

In order to remove objects from an object base instance, the GOOD transformation language has the node deletion operation.

Suppose that we are no longer interested in the info node corresponding to classical music in the hyper-media object base. Removing this information can be accomplished by the node deletion shown in Figure 14. Again this figure has two distinguishable parts. The first part is the source pattern which as always determines the relevant matchings. The second part consists of a single node (in double outline) specifying the nodes to be deleted. Figure 15 shows that part of the hyper-media object base affected by this node deletion. The info node with name *Classical Music* as well as the edges leaving it have been deleted. Notice also how as a result of this node deletion, the info node with name *Mozart* has become isolated in the object base. In general of course, one node deletion will remove several nodes.

The node deletion operation can also be useful in queries that involve negation. Assume that we want to tag all data info nodes that do not contain any sound data. This can be accomplished in a two step process involving a node addition and node deletion. The first step attaches to each data info node a node with label *No Sound*. The second step removes these tags from data info nodes containing sound data. The remaining tagged info nodes are those that do not contain sound data. So, negation can be thought of as a *macro*: see Section 4.1.

We are now ready for the formal definition of a 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 $\mathcal{S}$. ($\mathcal{J}$ will be called the source pattern of the node deletion*). Let $\mathbf{m}$ be a node in $\mathbf{M}$.*

The node deletion

$$\mathrm{ND}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathbf{m}]$$

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

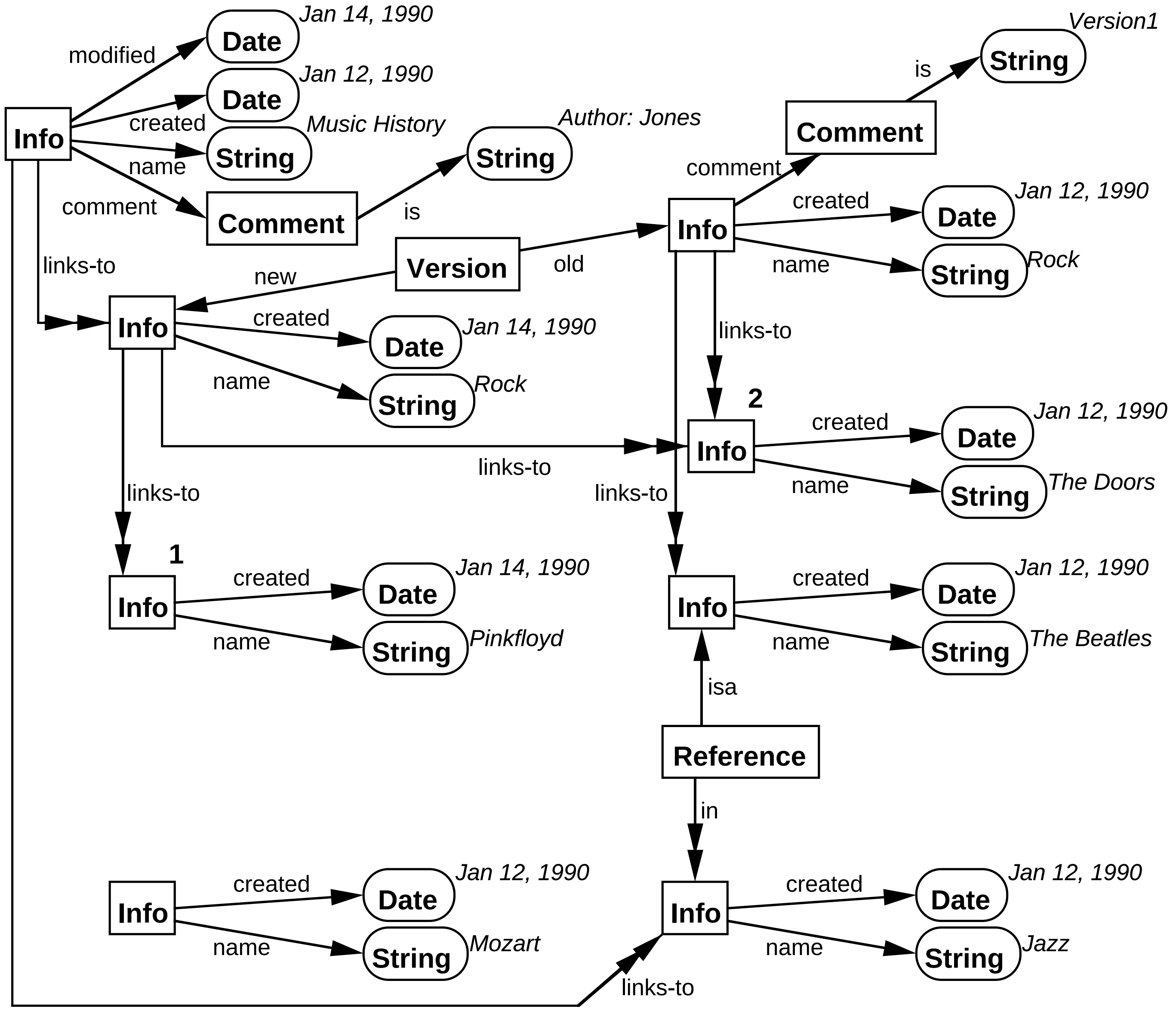

Figure 15: Result of the node deletion of Figure 14.

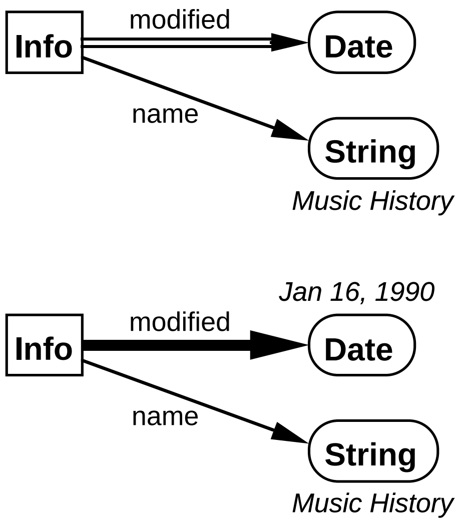

Figure 16: Example of an update through an edge deletion followed by an edge addition.

- *$\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ where $\mathbf{M}'$ is obtained by removing from $\mathbf{M}$ the node $\mathbf{m}$; $\mathbf{F}'$ is then obtained by removing from $\mathbf{F}$ all labeled edges involving $\mathbf{m}$;*
- *$\mathcal{S}'$ equals $\mathcal{S}$; and*
- *$\mathcal{I}'$ is the maximal instance over $\mathcal{S}'$ such that $\mathcal{I}'$ is a subinstance of $\mathcal{I}$, and such that for each matching i of $\mathcal{J}$ in $\mathcal{I}$, $i(\mathbf{m})$ is not a node of $\mathcal{I}'$.*

3.4 Edge deletion

In order to disassociate certain relationships between objects, the GOOD transformation language has the edge deletion operation.

Suppose we modified the info node with name *Music History* on Jan 16, 1990. Consequently, we need to update the *last-modified* property from *Jan 14, 1990* to *Jan 16, 1990*. This can be done in two steps. The first step, shown in Figure 16, involves the deletion of the edge with label *last-modified* from the info node with name *Music History* (notice how this edge is represented as a doubly outlined edge in the source pattern). The second operation, shown at the bottom of Figure 16, adds a new edge resulting in the intended update. Though not illustrated in this example, it should be clear that it is also possible to remove multivalued edges.

We are now ready for the formal definition of an 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 $\mathcal{S}$. ($\mathcal{J}$ will be called the source pattern of the edge deletion*). 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

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

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

- $\mathcal{J}' = (\mathbf{M}',\mathbf{F}')$ *where* $\mathbf{M}'$ *equals* $\mathbf{M}$ *and* $\mathbf{F}'$ *is obtained by removing from* $\mathbf{F}$ *the labeled edges* $(\mathbf{m}_1,\alpha_1,\mathbf{m}'_1),\ldots,(\mathbf{m}_n,\alpha_n,\mathbf{m}'_n)$*;*

- $\mathcal{S}'$ *equals* $\mathcal{S}$*; and*

- $\mathcal{I}'$ *is the maximal instance over* $\mathcal{S}'$ *such that* $\mathcal{I}'$ *is a subinstance of* $\mathcal{I}$*, and such that for each matching* 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 labeled edges of* $\mathcal{I}'$*.*

3.5 Abstraction

GOOD supports object identity. This means that objects exist independently of their properties. Sometimes, however, it is desirable to "abstract" over "duplicate" objects that share a same set of properties. The operation supporting this technique in GOOD is the abstraction operation. The abstraction creates a unique object for each equivalence class of duplicate objects; as such, it acts as a duplicate eliminator.

As an example, reconsider the hyper-media scheme specified in Figure 1. This scheme allows for the maintenance of different versions of info nodes. Now consider Figure 17. This figure displays a sub-instance of a hyper-media object base instance different from the one displayed in Figures 2 and 3. As can be seen, the info nodes pointed at by the version nodes share info nodes to which they are linked. In fact, in some cases, info nodes share the same set of other info nodes, as do for instance the first and second info nodes from the left. In order to "abstract" over info nodes which share the same set of nodes, consider Figure 18. This figure contains two node additions and an abstraction operation. The two node additions are used to tag the info nodes over which the abstraction will take place. An abstraction operation consists of three distinguishable parts. The first part (in solid lines) is the source pattern. The second part (in dashed lines) specifies the type of set equality (i.e., info nodes are grouped together if they are linked to the same set of info nodes). The third part (in bold lines) specifies the type of nodes and edges to be added as the result of the abstraction operation. The semantics of this operation is simply that for each group of info nodes being linked to the same set of info nodes, a new node with label *same-info* is introduced and linked to all the members of the group by edges with label *contains*. The result of this operation is shown in Figure 19.

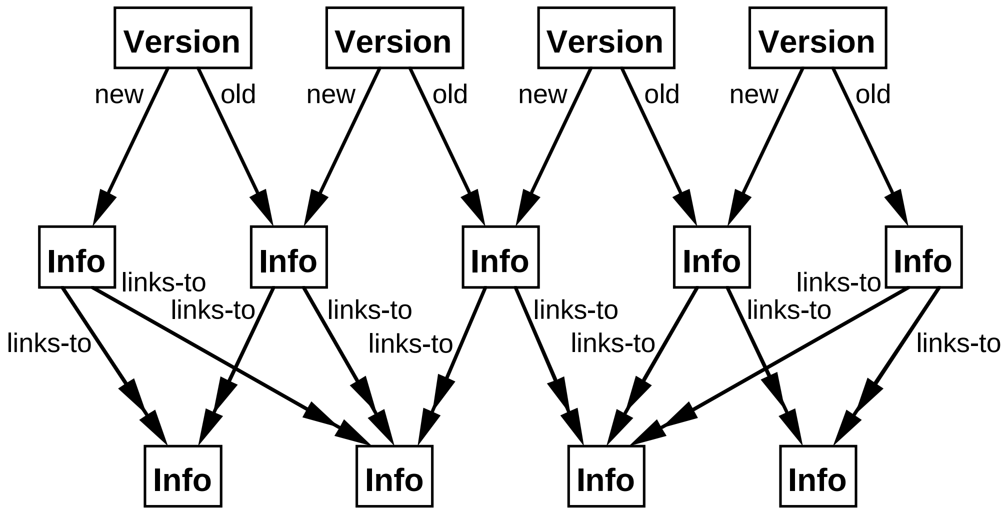

Figure 17: A sequence of versions of related information.

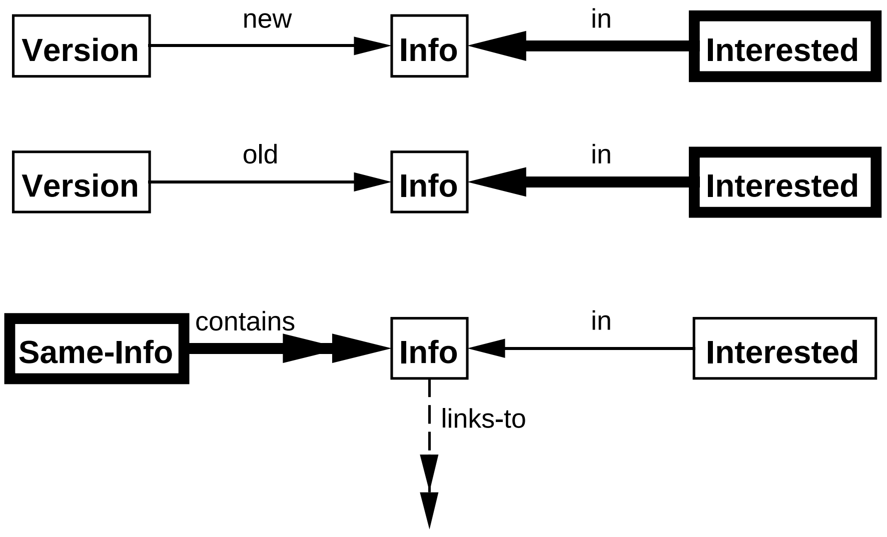

Figure 18: Example of an abstraction operation.

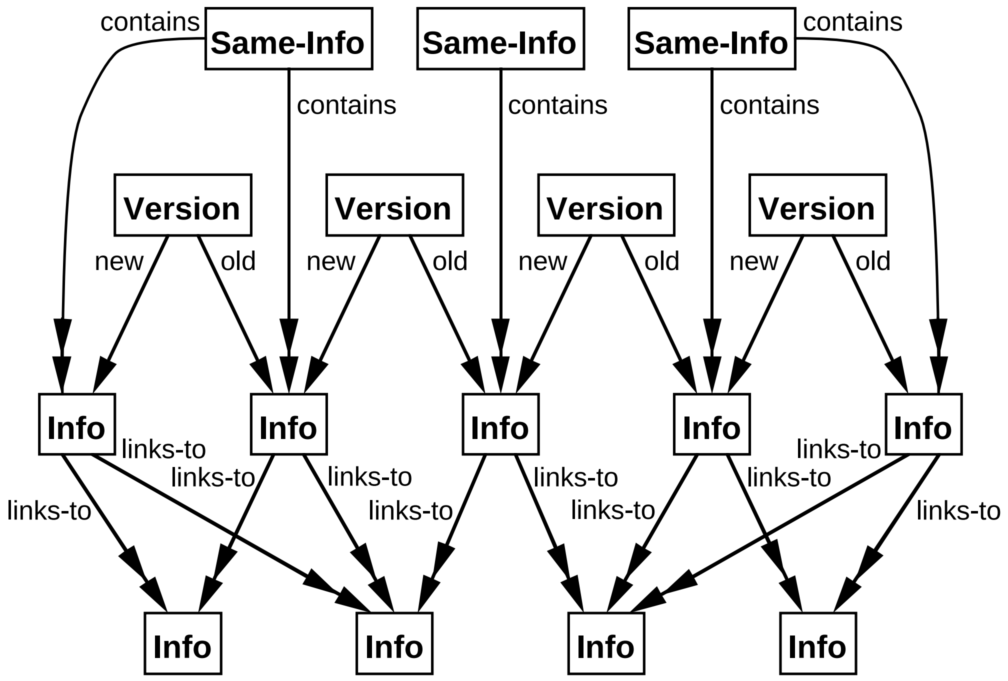

Figure 19: Result of the abstraction operation of Figure 18.

We are now ready for the formal definition of the abstraction operation.

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* $\mathcal{S}$*. (*$\mathcal{J}$ *will be called the* source pattern of the abstraction*). Let* $\mathbf{n}$ *be a node in* $\mathbf{M}$*. Let* K *be an object label, and let* α, β *be multivalued edge labels. Intuitively, the abstraction creates sets (labeled* K*). Each set contains all the objects* $\mathbf{n}$ *that match the pattern* $\mathcal{J}$ *and that have the same* α *properties.*

More formally, the abstraction

$$\mathrm{AB}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathbf{n}, K, \alpha, \beta]$$

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

- $\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ *where* $\mathbf{M}'$ *is obtained by adding to* $\mathbf{M}$ *a new node* $\mathbf{m}$ *with label* K*;* $\mathbf{F}'$ *is then obtained by adding to* $\mathbf{F}$ *the labeled multivalued edge* $(\mathbf{m}, \beta, \mathbf{n})$*;*
- $\mathcal{S}'$ *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* $\mathcal{S}'$ *such that*

  1. $\mathcal{I}$ *is a subinstance of* $\mathcal{I}'$*;*

2. *for each matching i of $\mathcal{J}$ in $\mathcal{I}$, there exists a K-labeled node* $\mathbf{p}$ *in* $\mathcal{I}'$ *such that* $(\mathbf{p}, \beta, \mathbf{m})$ *is an edge of* $\mathcal{I}'$ *if and only if the sets of nodes* $\{\mathbf{r} : (\mathbf{m}, \alpha, \mathbf{r}) \in \mathbf{E}\}$ *and* $\{\mathbf{r} : (i(\mathbf{n}), \alpha, \mathbf{r}) \in \mathbf{E}\}$ *are equal; and*

3. *each edge in* $\mathcal{I}'$ *leaving a node of* $\mathcal{I}$ *is also an edge of* $\mathcal{I}$.

Observe that abstraction is always well defined. The third condition captures the essence of the concept of abstraction.

The reader may wonder why we define abstractions only over one single multivalued property α. It could indeed be useful to group together objects that agree on a *set* of functional or multivalued properties. However, it can be shown that abstraction over functional properties is expressible using the other GOOD operations introduced in this section. Furthermore, abstraction over multiple properties can always be reduced to abstraction over one single property. More details on the expressive power of abstraction are given in [32].

3.6 Methods

As in any high-level programming language, it is useful to incorporate in the GOOD transformation language, a programming construct to allow the grouping of a sequence of other operations. Furthermore, the object-oriented approach in software engineering advocates the principle of *encapsulation*, where code is associated to objects of a given *receiver* class. All this is supported in GOOD through *methods*.

For example, consider a method 'Update' for updating the last-modified date of an info node. It takes a Date object as parameter, and a call to the method can be received by an info node object. This information: the name, the labels of the parameters and the label of the receiver, is given by the *method specification*, shown at the top of Figure 20. Now, the *body* of the method, shown in the remainder of Figure 20, consists of an edge deletion, deleting the old information, followed by an edge addition, inserting the new information. Note how the diamond-shaped method node is used to bind pattern nodes to the formal receiver and formal parameters in the method body. Having written the body, we can finally *call* the method for all info nodes satisfying a certain pattern. This is shown in Figure 21, where the last-modified date of each info node with name 'Music History' is updated to 'Jan 16, 1990'. Again a diamond-shaped method node, now in bold, is used to bind pattern nodes to the actual receiver and actual parameters of the method call.

Methods can also be used to specify recursive processes. Suppose that we want to remove all the old versions of an info node. This requires a kind of transitive closure of the Version relationship, which is impossible using only the basic five operations of the GOOD transformation language. We next show how methods can be used to overcome this problem. Consider the method in Figure 22. The method specification introduces

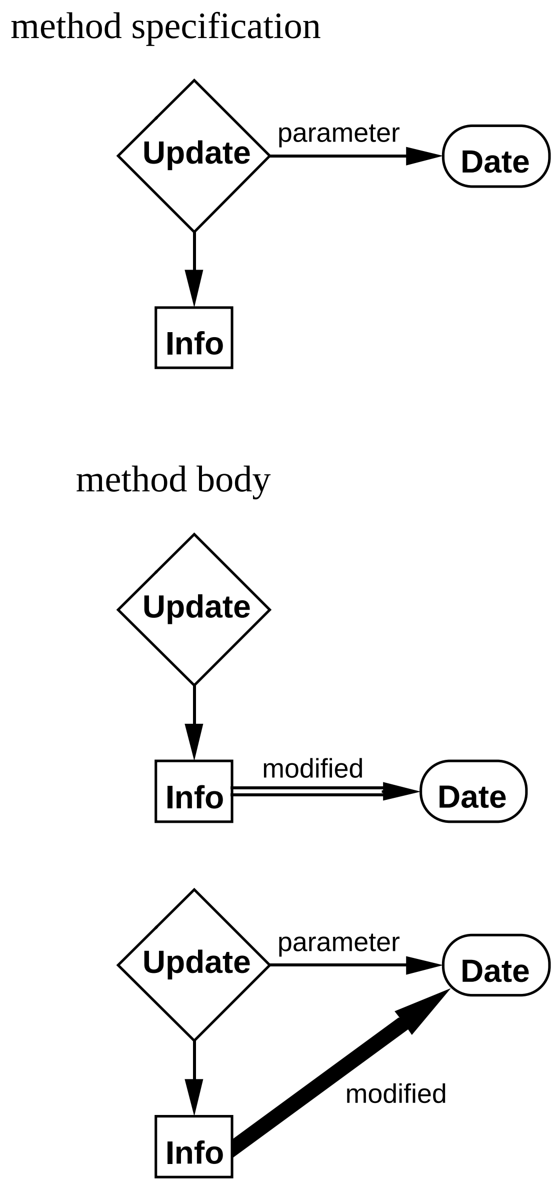

Figure 20: A method to change the last modification date of an info node.

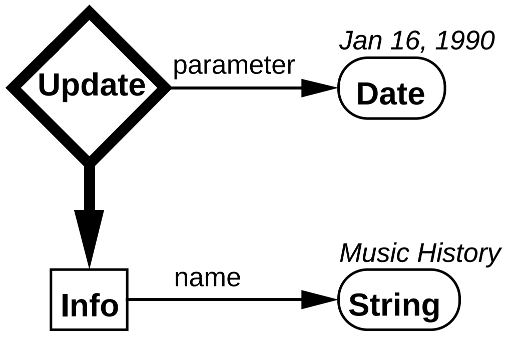

Figure 21: A method call to change the last modification date of *Music History* info nodes.

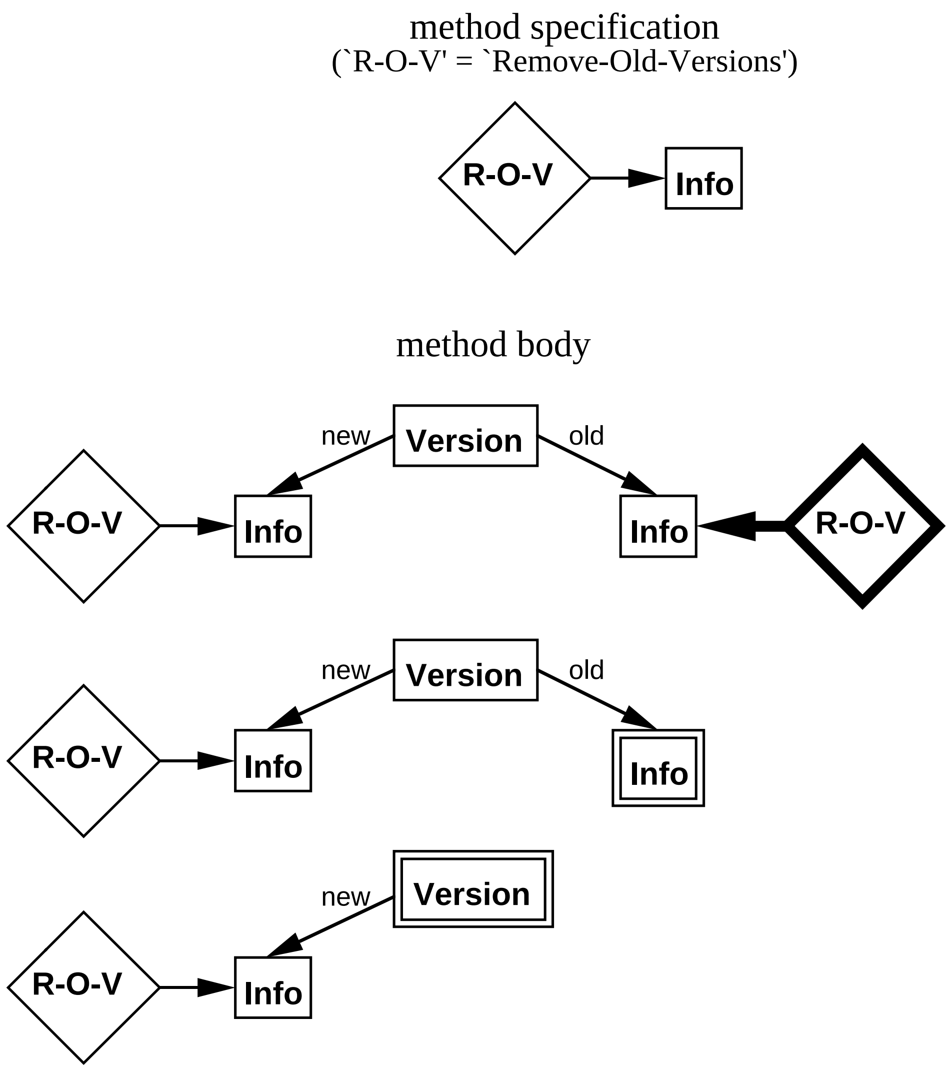

Figure 22: An example of a recursive method to delete old versions of an info node.

the method *Remove Old Versions* with as receivers info nodes. The method body consists of three operations. The first operation involves a recursive call (bold diamond-shaped node) which removes all the old versions of the current receiver (pointed at by the regular diamond-shaped node). Notice that the recursion halts when a receiver info node does not have a previous version. The bottom two operations actually perform the appropriate node deletions. First the version node directly associated with the receiver is deleted. Then the no longer useful version node is removed.

Besides the method specification, body, and call, there is a fourth basic part of our method mechanism: the method *interface*. Method interfaces allow the user of a method to compute the scheme resulting from the method without any knowledge of the method's body. Concretely, an interface is a (usually small) scheme which describes the effect of the method at the scheme level. If, in order to compute the desired effect of the method,

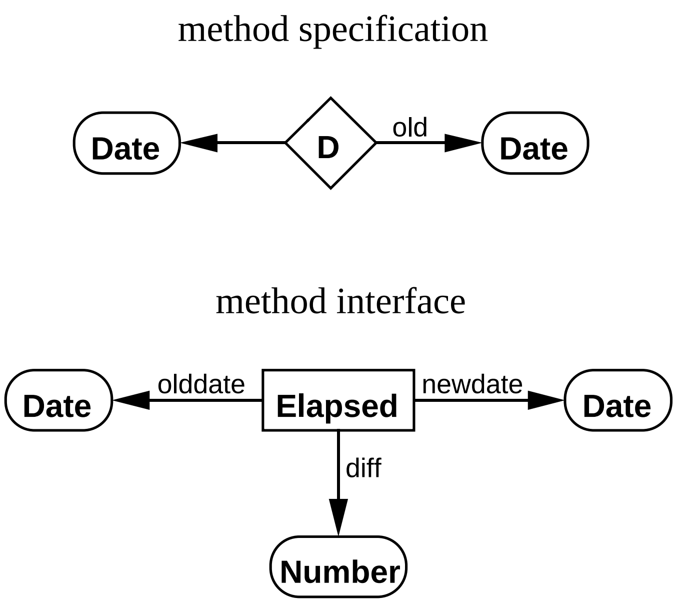

Figure 23: Specification and interface of a method to compute the number of days elapsed between two dates.

intermediate operations in the method body introduce some temporary nodes and edges, which are irrelevant to the final result, the method interface will automatically filter them out.

To illustrate the interface mechanism, consider a method to compute the number of days between two given dates, the specification and interface of which are shown in Figure 23. Given the user knows the meaning of the labels used in Figure 23, he can now employ the method D without having to have any knowledge whatsoever about the method body. In other words, the method interface serves to hide implementation details for the user.

Using the method D, it is now easy to write a method that, for each info node, computes the number of days elapsed between its creation and its last modification (Figures 24 and 25). Observe that the *Elapsed* nodes, introduced as an effect of calling the method D, will *not* appear in the resulting instance, even though they are not deleted in the method body, as these nodes do neither occur in the original scheme nor in the method interface.

We are now ready for the formal definition of the method concept.

A GOOD method is a named *procedure. It has* receiver *and* parameter *labels, a* specification*, a* body*, and an* interface*. Let $\mathcal{S} = (OL, POL, FEL, MEL, \mathcal{P})$ be an object base scheme.*

The method specification contains the method's name and parameter types. Formally, the method specification *of a method $\mathcal{M}$ is a pair $(s_{\mathcal{M}}, R_{\mathcal{M}})$, where $s_{\mathcal{M}}$ is a total function, $s_{\mathcal{M}}\colon L_{\mathcal{M}} \rightarrow OL \cup POL$, with $L_{\mathcal{M}}$ a finite (possibly empty) set of functional edge labels. $s_{\mathcal{M}}$*

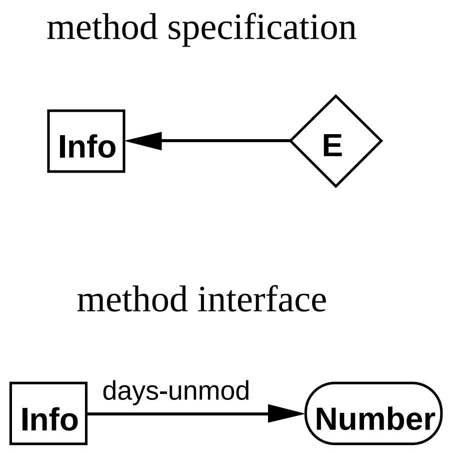

Figure 24: Specification and interface of a method to compute the number of days elapsed between creation and last modification of an info node.

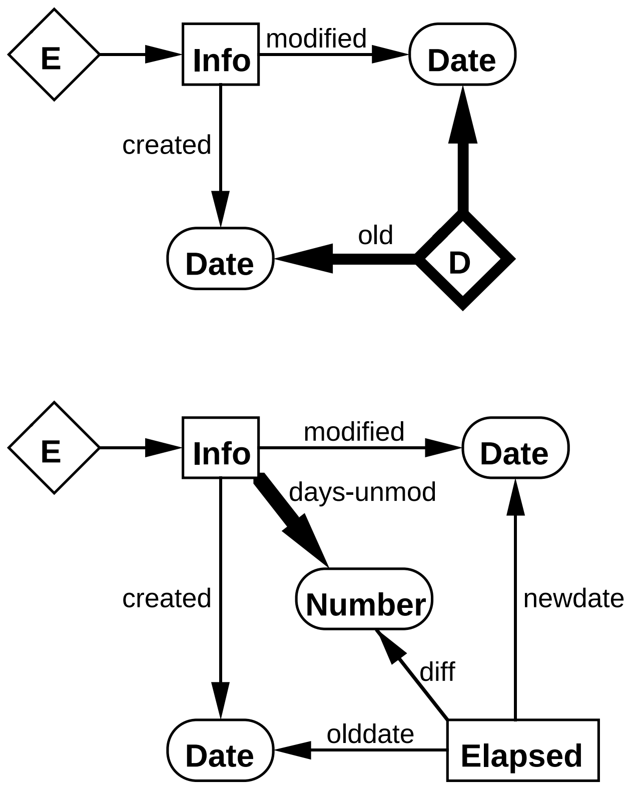

Figure 25: Body of a method to compute the number of days elapsed between creation and last modification of an info node.

associates with each of these parameter labels a node label. $R_{\mathcal{M}} \in OL \cup POL$ *is the node label of the receiver. Graphically,* $\mathcal{M}$ *is represented by a diamond-shaped node that is labeled by* $\mathcal{M}$*, with a labeled outgoing edge for each label* $\beta \in L_{\mathcal{M}}$ *to a node labeled by* $s_{\mathcal{M}}(\beta)$*, and an unlabeled outgoing edge to a node labeled by* $R_{\mathcal{M}}$*.*

The method body *specifies the implementation of the method. Formally, the* method body $B_{\mathcal{M}}$ *of a method* $\mathcal{M}$ *is a sequence of* parameterized operations. *Parameterized operations are normal operations (i.e.,* NA*,* ND*,* EA*,* ED*,* AB *or* MC *(method call, see further)) or normal operations where the source pattern* $\mathcal{J}$ *is augmented with one diamond-shaped node labeled by* $\mathcal{M}$*, called the* $\mathcal{M}$*-head-node, and with edges leaving that node. At most one edge for each label* β *of* $L_{\mathcal{M}}$ *can leave the* $\mathcal{M}$*-head-node. It has to point to a node labeled by* $s_{\mathcal{M}}(\beta)$*. Furthermore, there can be an unlabeled outgoing edge to a node labeled by* $R_{\mathcal{M}}$*. No other edges can leave the* $\mathcal{M}$*-head-node.*

The method interface $\mathcal{C}_{\mathcal{M}}$ *is formally a scheme.*

The method call *is the operation that invokes the execution of the method body in a context specified by a pattern and actual parameters. Formally, let* $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ *be an object base instance and* $\mathcal{J} = (\mathbf{N}', \mathbf{F})$ *a pattern over* $\mathcal{S}$*. Let* $\mathcal{M} = (s_{\mathcal{M}}, R_{\mathcal{M}})$ *be the specification of a method over* $\mathcal{S}$*,* g *be a total function,* $g\colon L_{\mathcal{M}} \rightarrow \mathbf{N}'$ *where* $g(\beta)$ *must have the label* $s_{\mathcal{M}}(\beta)$*, let* $\mathbf{n}$ *be a node in* $\mathcal{J}$ *with node label* $R_{\mathcal{M}}$*. The* method call $\mathrm{MC}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathcal{M}, g, \mathbf{n}]$ *is graphically represented by the pattern* $\mathcal{J}$ *augmented with a bold diamond shaped node, labeled* $\mathcal{M}$*, and a bold edge for each* $\beta \in L_{\mathcal{M}}$ *to the node* $g(\beta)$ *and a bold edge to* $\mathbf{n}$*.*

The semantics of the method call is then that the steps in the body of the method are executed consecutively, but *only for those nodes in the instance under consideration that match the nodes in the pattern to which the method parameters point and only with the actual values of the parameters.*

Formally, the method call

$$\mathrm{MC}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathcal{M}, g, \mathbf{n}]$$

results in a new scheme $\mathcal{S}'$ *and a new instance* $\mathcal{I}'$ *over* $\mathcal{S}'$ *defined as follows. Consider the node addition* $\mathrm{NA}[\mathcal{J}, \mathcal{S}, \mathcal{I}, K, \{(\beta, g(\beta)) \mid \beta \in L_M\} \cup \{(\beta_{\mathbf{n}}, \mathbf{n})\}] = (\mathcal{J}_0, \mathcal{S}_0, \mathcal{I}_0)$*. Let* $(PO_1, PO_2, \ldots, PO_k)$ *be the method body. We define for every parameterized operation* PO_i *an operation* $OPER_i$ *as follows:*

- *If* PO_i *is a normal operation then* $OPER_i$ *is the same operation as* PO_i*, except that an isolated node labeled* K *is added to the source pattern of* $OPER_i$*;*

- *If the source pattern of* PO_i *contains a diamond-shaped node labeled by* M*, then* $OPER_i$ *is the same operation as* PO_i*, except that this diamond-shaped node is substituted by a rectangular-shaped node labeled by* K*.*

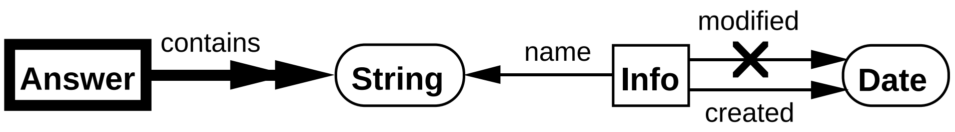

Figure 26: Expressing absence of edges.

Let now $(\mathcal{S}_i, \mathcal{I}_i)$ *be the result of executing* PO_i *on* $(\mathcal{S}_{i-1}, \mathcal{I}_{i-1})$*. The execution of these operations eventually results in* $(\mathcal{S}_k, \mathcal{I}_k)$*. Let* $\mathrm{ND}[\mathcal{J}_K, \mathcal{S}_k, \mathcal{I}_k, \mathbf{m}] = (\mathcal{J}_{k+1}, \mathcal{S}_{k+1}, \mathcal{I}_{k+1})$*, where* $\mathcal{J}_K$ *is the pattern that has no edges and only contains one node* **m** *labeled* K*. Then* $\mathcal{S}'$ *is defined as the union*[3] *of* $\mathcal{S}$ *and* $\mathcal{C}_{\mathcal{M}}$*, and* $\mathcal{I}'$ *is defined as* $\mathcal{I}_{k+1}$ *restricted*[4] *to* $\mathcal{S}'$*.*

4 Other features of GOOD

In this section, we discuss the following additional aspects of the GOOD model: macros, object-orientation, and computational completeness.

4.1 Macros

A variety of additional graphical operation can be added to the GOOD language. They allow for more succinct expression of certain frequently occurring data manipulations; however, they do not increase the expressive power of the language. Hence, they can be thought of as macros. In this subsection, we consider a number of possible macros.

Negation. The pattern matching technique checks for the *presence* of nodes and edges in a particular combination. For some transformations, however, we need the *absence* of nodes or edges. Consider for instance the query: "Give the set of the names of the info nodes with a creation date that is different from its last-modified date". This query is shown in Figure 26. The crossed edge indicates that we are interested in the patterns that have *no* last-modified edge between the indicated nodes (similarly one can consider patterns with crossed nodes). As already suggested in Section 3.3, the general technique to simulate patterns with a crossed part in GOOD utilizes deletions. Figure 27 simulates the query of Figure 26. First, intermediate nodes are created for every matching of the non-crossed part of the pattern. Then the intermediate nodes are deleted that are associated to an matching that can be enlarged to the complete pattern. The intermediate nodes that are left represent the desired matching.

[3] I.e., the smallest scheme of which both $\mathcal{S}$ and $\mathcal{C}_{\mathcal{M}}$ are subgraphs (recall that $\mathcal{C}_{\mathcal{M}}$ is the method interface.)
[4] I.e., the largest subinstance of $\mathcal{I}_{k+1}$ that is an instance over $\mathcal{S}'$.

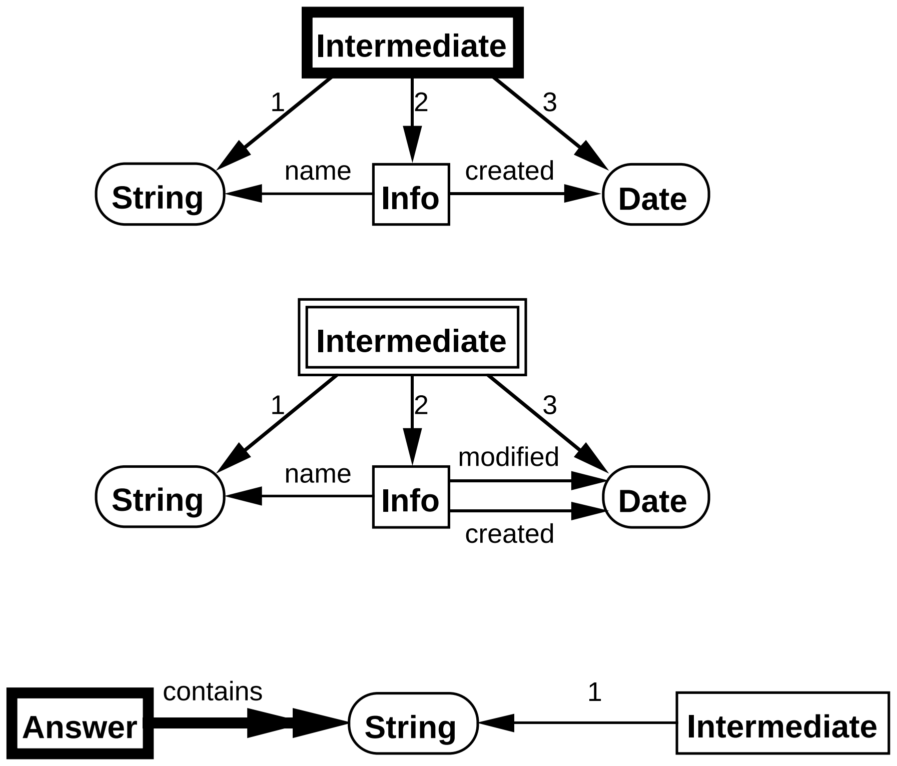

Figure 27: Simulation of negation in GOOD.

Additional predicates on printable objects. In practice, queries need more involved conditions on printable objects than merely testing for equality. For example, checking whether certain values are in a given range. As an example consider the request to determine the info nodes created between January 1, 1990 and January 31, 1990. A straightforward extension of the GOOD model allowing the specification of extra conditions (possible using external functions) applied to printable objects (e.g., in the style of QBE's *condition boxes* [38]) would allow one to handle this query.

Recursive addition operations. Suppose that we want to compute the transitive closure of the *links-to* property. Concretely, we want to add an edge labeled *rec-links-to* between any two info nodes that are connected by *links-to* edges. The first operation of Figure 28 is a standard edge addition, specifying the direct links. The second operation is a *recursive* edge addition. The starred edge indicates that the edge addition is repeated as long as new *rec-links-to* edges can be added. Similarly, one can consider recursive node addition. Note however that this can result in an infinite sequence of node additions. As already suggested in Section 3.6, the general technique to simulate recursive operations in GOOD utilizes recursive method calls. The method in Figure 29 simulates the recursive edge addition of Figure 28. The first operation in the method body uses the given pat-

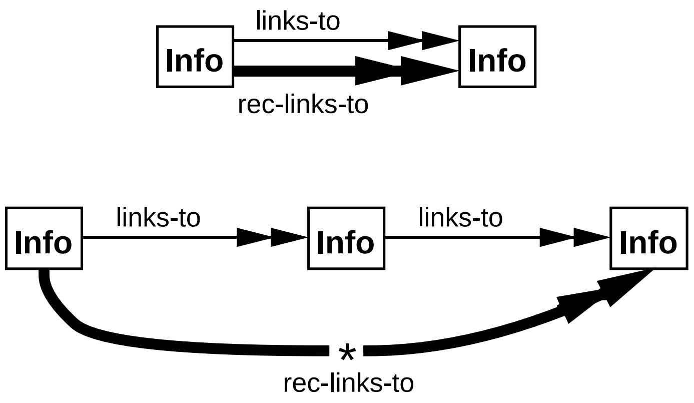

Figure 28: Computing transitive closure using recursive edge addition.

tern with corresponding method parameters, and performs the "underlying" non-starred operation. The second operation in the body calls the method recursively. The pattern is augmented with a crossed part that corresponds to the starred part of the recursive operation: this expresses the stopping condition for the recursion.

4.2 Object-oriented aspects of GOOD

In this subsection, we show how GOOD can account for a number of object-oriented features as described in [3].

Complex objects and object-identity. *Complex objects* [3] are typically built from printable objects according to certain object constructors, such as tuples, sets and lists. These structures all have a natural graph-based representation. Therefore, GOOD is a natural model for working with complex objects.

The notion of *object-identity* refers to the existence of objects in the database independent of their associated properties, so that objects can be shared and updated independently. As stressed from the outset, object-identity is a basic feature of GOOD.

Encapsulation. Encapsulation means that an object can be accessed only through the methods defined in its class. The precise representation of the data and implementation of the methods is invisible. It is customary in this respect to divide the methods into two categories: those which only retrieve certain properties of the object and have no side effects, and those which do have side effects. Properties of objects are modeled in GOOD as edges in the instance graph; general methods with side effects are provided by the GOOD method mechanism.

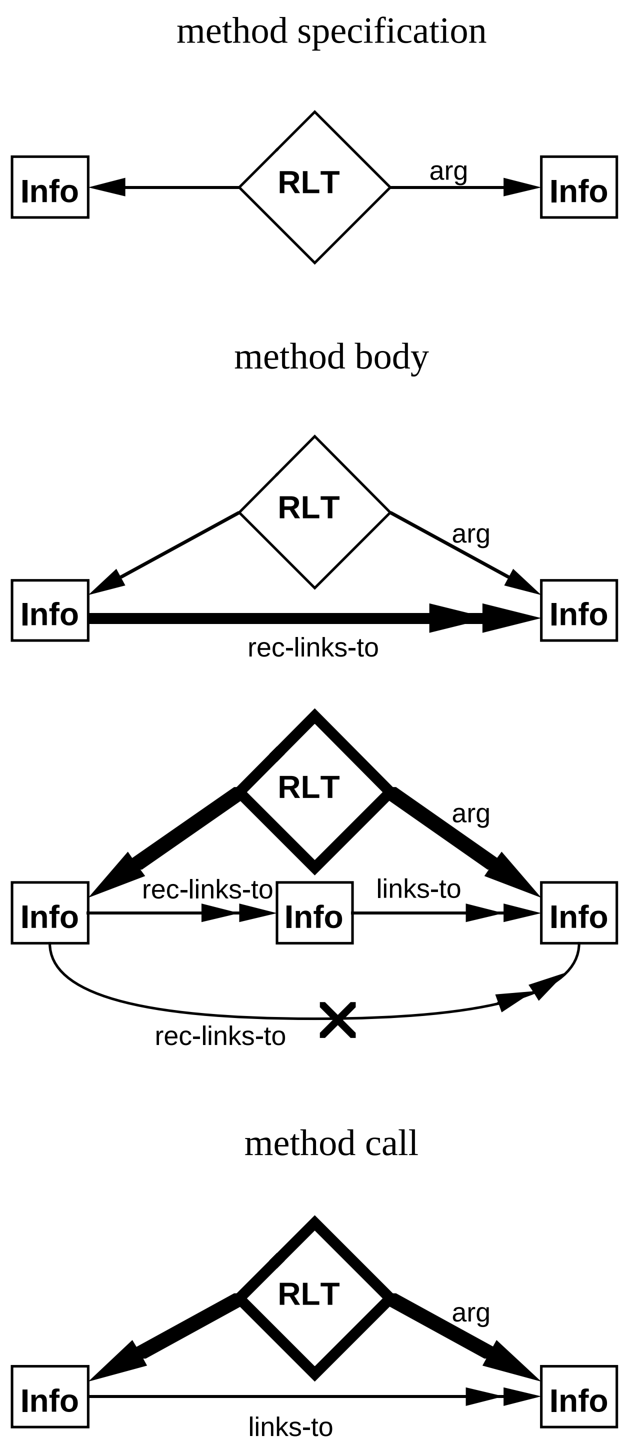

Figure 29: Simulation of recursion in GOOD.

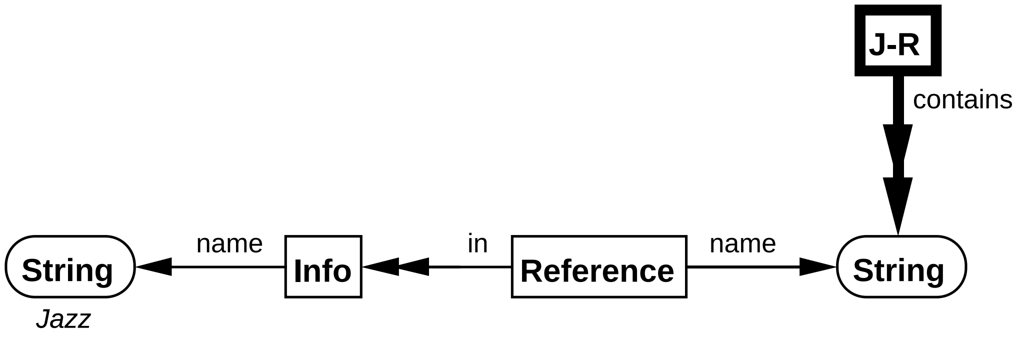

Figure 30: A GOOD query utilizing inheritance.

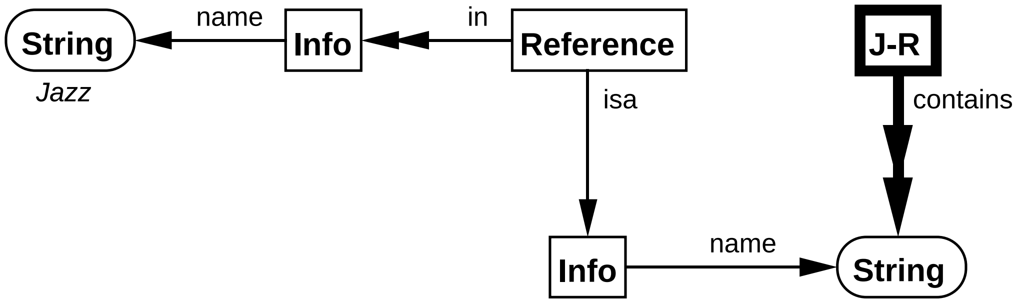

Figure 31: Simulation of inheritance in GOOD.

Inheritance. Object-oriented databases support some form of *inheritance*, i.e., the ability to define new classes as subclasses of existing ones, by organizing the classes in a *class hierarchy*. We can modestly extend the GOOD model and let functional edge labels support the notion of subclass. Those functional edges in the scheme graph we wish to interpret as subclass edges must of course somehow be marked. We will also assume that the subclass edges do not form a cycle.

For example, we can consider the *isa* edges in the hyper-media object base of Figures 1–3 between *Reference* and *Info* nodes as subclass edges. The effect to the user is the same as if all properties of info objects were also attached to the corresponding reference objects. The user can now apply *Info* operations directly to *Reference* objects. For example, in order to obtain all references to Jazz, the user may specify the query of Figure 30. Since *name* is not a property of *Reference*, GOOD will translate this query internally into the query of Figure 31. Similarly, a method can be called on objects belonging to subclasses of the method's specified receiver and parameter classes. Note that, while designing a GOOD scheme, the user must be very careful to define the *isa*-links unambiguously.

In other words, using inheritance in formulating GOOD queries comes down to working in a virtual instance obtained by explicitly adding the properties of the target nodes of an *isa*-link to the source nodes as well. Clearly, this transformation can be computed by a number of consecutive edge additions. Hence, the use of inheritance in patterns is but a *macro* in the sense of the previous subsection. In fact one can argue that *isa*-links actually define a *view* of the object base the user can work with to formulate his or her queries.

4.3 Computational completeness

Programs in GOOD are built from the five basic operations, node (edge) addition, node (edge) deletion and abstraction, together with method construction. In this subsection we briefly discuss the computational power of the GOOD language.

When we restrict the language to only node and edge additions and deletions, we obtain a language which is *relationally complete* in the well-known sense proposed by Codd [5]. Concretely, suppose we represent a relation R with attributes A_1, A_2, A_3 with domains D_1, D_2, D_3 as a class R with functional edges labeled A_1, A_2, A_3 to printable classes D_1, D_2, D_3. Tuples of R are represented by objects of this class. Then using this simulation of relations and tuples as classes and objects, we invite the reader to convince himself that every relation computable in the relational algebra is also computable in the restricted GOOD language (see also [16]).

By adding abstraction, one can moreover simulate the nested relational algebra [28]. Nested relations are represented in an analogous manner as standard relations, now using also multivalued edges. The abstraction operation is needed in this case to obtain "faithful" simulations of relation-valued attributes, meaning that duplicate relations can be eliminated (see also [32]). Again, the details of the simulation are left to the reader.

The full language with methods is sufficiently strong to simulate arbitrary Turing Machines; this can be shown using well-known techniques (e.g., [34]).

Finally, on the most general level, one may ask exactly which graph transformations (i.e., computable mappings from graphs to graphs) can be expressed in GOOD. This question was addressed in [33]; informally speaking, it was shown there that GOOD can express all isomorphism-preserving transformations for which newly created objects can be effectively "constructed".

5 Concluding remarks

We end the paper with some concluding remarks.

The GOOD transformation language is reminiscent of graph grammars [10]. An application of a graph grammar production rewrites a matching of a pattern by another pattern. Similarly, the basic operations node (edge) addition (deletion) rewrite matchings of a pattern by adding or deleting nodes or edges. (The fifth basic operation, abstraction, works more global and does not fit this description.) There are however important differences between the GOOD approach and graph grammars. First, in graph grammars, the main objectives are to find convenient, general formalisms for specifying rewritings of subgraphs; this involves solving difficult problems such as determining how the rewritten subgraph has to be "glued" into the original graph. In GOOD, we wish to avoid these (not yet completely resolved) problems and employ only the simplest rewritings (node and edge addition and deletion). A second important difference is that the operational semantics of (graph) grammar derivations is non-deterministic, both in the choice of the production to be applied as in the choice of the particular matching of the corresponding pattern to be rewritten. In GOOD, basic operations are applied in a predetermined order (possibly within method executions), and, importantly, work on *every* matching of the pattern, in parallel. This is in line with the more set-oriented processing nature of database systems.

Although GOOD programs are written in a procedural way, the basic operations node (edge) addition (deletion) have a partly declarative nature. Indeed, the pattern of such an operation can be seen as the (declarative) condition part of a rule, while the bold or outlined part corresponds to a rule's action (in this case the addition or deletion of nodes or edges). This simple mechanism for visualization of rules can provide a basis for the development of graph-based, rule-based, object-oriented database languages [24].

The GOOD system is currently being implemented at the University of Antwerp [8]. Fundamental design concepts for graph-based database user interfaces in the spirit of GOOD are discussed in [2]. A concrete database user interface for GOOD has been developed. The interface provides graphical tools to specify patterns and GOOD programs, as well as tools for pattern-directed browsing. The graphical representation of the object base scheme can be customized. More details can be found in [13]. A prototype of the actual data management is implemented on top of a relational system. Classes are stored as relations with attributes for the object identifier and the functional properties. Multi-valued edges are stored as binary relations. The set of all matchings of the pattern of a GOOD operation is expressed as an SQL query. The actual transformation is performed using SQL's update capabilities. In this way, GOOD programs (including methods) are interpreted by C programs with embedded SQL statements.

At Indiana University, an alternative approach to implementing the GOOD system is explored [27]. There, a binary relational model, called the *Tarski Data Model*, is used to store and compute with GOOD databases. The model includes its own (binary) relational algebra, which is inspired by Tarski's work.

Acknowledgment

We wish to thank Marc Andries, Latha Colby, Marc Gemis, Peter Peelman, Ed Robertson, Vijay Sarathy, Larry Saxton, and Inge Thyssens for helpful discussions. We also wish to

thank the referees for their constructive criticism.

References

[1] S. Abiteboul and R. Hull. Data functions, datalog and negation. In H. Boral and P.A. Larson, editors, *1988 Proceedings SIGMOD International Conference on Management of Data*, pages 143–153. ACM Press, 1988.

[2] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. Concepts for graph-oriented object manipulation. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology—EDBT'92*, volume 580 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 1992.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proceedings 1st International Conference on Deductive and Object-Oriented Databases*, pages 40–57. Elsevier Science Publishers, 1989.

[4] D. Bryce and R. Hull. SNAP: A graphics-based schema manager. In *Proceedings of the International Conference on Data Engineering*, pages 151–164, 1986.

[5] E. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, 1972.

[6] J. Conklin. Hypertext: An introduction and survey. *Computer*, 20(9):17–41, 1987.

[7] M. Consens and A. Mendelzon. GraphLog: A visual formalism for real life recursion. In PODS [26], pages 404–416.

[8] M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. GOOD: A graph-oriented object database system. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data, ACM SIGMOD Record* 22(2):505–510, 1993.

[9] C. Davis, S. Jajodia, P. Ng, and R. Yeh, editors. *Entity-Relationship Approach to Software Engineering: Proceedings of the International Conference on Entity-Relationship Approach.* North-Holland, 1983.

[10] H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Graph-Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[11] R. Elmasri and J. Larson. A graphical query facility for ER databases. In J. Liu, editor, *Proceedings of the Fourth International Conference on Entity-Relationship Approach*, pages 236–245. IEEE Computer Society Press, 1985.

[12] D. Fogg. Lessons from a "Living In a Database" graphical query interface. In B. Yormark, editor, *Proceedings of SIGMOD 84 Annual Meeting*, volume 14:2 of *SIGMOD Record*, pages 100–106. ACM Press, 1984.

[13] M. Gemis, J. Paredaens, and I. Thyssens. A visual database management interface based on GOOD. Technical Report 92-04, UIA Dept. Math. & Comp. Science, 1992. To appear in *Proceedings International Workshop on Interfaces to Database Systems*, Springer-Verlag.

[14] K.J. Goldman, S.A. Goldman, P.C. Kanellakis, and S.B. Zdonik. ISIS: Interface for a semantic information system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, volume 14:4 of *SIGMOD Record*, pages 328–342. ACM Press, 1985.

[15] M. Gyssens, J. Paredaens, and D. Van Gucht. A grammar-based approach towards unifying hierarchical data models. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, volume 18:2 of *SIGMOD Record*, pages 263–272. ACM Press, 1989.

[16] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In PODS [26], pages 417–424.

[17] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model for database end-user interfaces. In H. Garcia-Molina and H.V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, volume 19:2 of *SIGMOD Record*, pages 24–33. ACM Press, 1990.

[18] S. Heiler and A. Rosenthal. G-WHIZ, a visual interface for the functional model with recursion. In *Proceedings 11th International Conference on VLDB*, pages 209–218, 1985.

[19] R. Hull and R. King. Semantic database modelling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[20] W. Kim and F.H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* Frontier Series. ACM Press, Addison-Wesley, 1989.

[21] R. King and S. Melville. The semantics-knowledgeable interface. In *Proceedings of the 10th VLDB Conference*, pages 30–37, 1984.

[22] A. Klug. Calculating constraints on relational expressions. *ACM Transactions on Database Systems*, 5(3):260–290, 1980.

[23] A. Motro, A. D'Atri, and L. Tarantino. The design of KIVIEW an object-oriented browser. In *Proceedings 2nd International Conference on Expert Database Systems*, pages 73–106, 1988.

[24] J. Paredaens, P. Peelman, and L. Tanca. G-Log: A declarative graphical query language. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 1991.

[25] J. Peckham and F. Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3):153–190, 1988.

[26] *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*. ACM Press, 1990.

[27] V.M. Sarathy, L.V. Saxton, and D. Van Gucht. Algebraic Foundation and Optimization for Object Based Query Languages. *Proceedings Ninth International Conference on Data Engineering*, pages 81–90. IEEE Computer Society Press, 1993.

[28] H.-J. Schek and M.H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[29] D. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 16(10):140–173, 1981.

[30] M. Stonebraker, editor. *Readings in Database Systems*. Morgan Kaufmann, 1988.

[31] Ursprung and Zehnder. HIQUEL: An interactive query language to define and use hierarchies. In Davis et al. [9], pages 299–314.

[32] J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 291–299. ACM Press, 1991.

[33] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating query languages. In *Proceedings 33rd Symposium on Foundations of Computer Science*, pages 372–379. IEEE Computer Society Press, 1992.

[34] D. Varvel and L. Shapiro. The computational completeness of extended database query languages. *IEEE Transactions on Software Engineering*, 15(5):632–637, 1989.

[35] H.K. Wong and I. Kuo. GUIDE: A graphical user interface for database exploration. In *Proceedings 8th International Conference on VLDB*, pages 22–32, 1982.

[36] S.B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1989.

[37] Zhang and Mendelzon. A graphical query language for entity-relationship databases. In Davis et al. [9], pages 441–448.

[38] M. Zloof. Query-by-example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

Index terms

database models

query languages

graph transformations

object-oriented databases

user interfaces

Figure captions

Footnotes

[*]Preliminary versions of this paper were presented at the 9th ACM Symposium on Principles of Database Systems [16] and the 1990 ACM SIGMOD International Conference on Management of Data [17]. This work was partially supported by the DPWB of Belgium under program IT/IF/13.

[†]University of Limburg, Dept. WNI, Universitaire Campus, B-3590 Diepenbeek, Belgium. E-mail: gysm@bdiluc01.bitnet.

[‡]University of Antwerp (UIA), Dept. Math. & Comp. Science, Universiteitsplein 1, B-2610 Antwerp, Belgium. E-mail: Paredaens: pareda@wins.uia.ac.be; Van den Bussche: vdbuss@wins.uia.ac.be. Jan Van den Bussche is a Research Assistant of the NFWO.

[§]Indiana University, Comp. Science Department, Bloomington, IN 47405-4101, USA. E-mail: vgucht@cs.indiana.edu.

[1]We should note that we do not intend to present this typically large and complex graph as such to the user. The GOOD transformation language, to be introduced in the next section, provides tractable primitives for manipulating and visualizing relevant parts of the instance graph.

[2]Subscheme and subinstance are defined with respect to set inclusion.

[3]I.e., the smallest scheme of which both $\mathcal{S}$ and $\mathcal{C}_{\mathcal{M}}$ are subgraphs (recall that $\mathcal{C}_{\mathcal{M}}$ is the method interface.)

[4]I.e., the largest subinstance of $\mathcal{I}_{k+1}$ that is an instance over $\mathcal{S}'$.

Preferred address for all correspondence

Dirk Van Gucht

Indiana University

Computer Science Dept.

Bloomington

Indiana 47405-4101

telephone: 1-812-855 6429

fax: 1-812-855 4829

e-mail: `vgucht@cs.indiana.edu`