TECHNICAL REPORT NO. 327

A Graph-Oriented Object Database Model

by

Marc Gyssens, University of Limburg
Jan Paredaens and Jan Van den Bussche, University of Antwerp
and
Dirk Van Gucht, Indiana University

March 1991

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

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. The scheme as well as the instance of an object database are represented by graphs. The data manipulation is expressed by graph transformations. These graph transformations are described by five primitive operations with a natural semantics. Based on these operations, a number of powerful macros are presented. It is illustrated how *GOOD* can be used as a model for an object-oriented database user interface.

Index terms: data model, graph, rewriting system, transformation language, object-oriented database.

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 extended architectures [10, 32, 36] and object-oriented databases [6, 7, 23, 36].

Along with this trend, the need for better and easier-to-use database end-user interfaces has been stressed [32, 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 [7, 21, 23, 29]. Their usage in data manipulation languages, however, is far more sparse. To deal with the language component, typically schemes in semantic and object-oriented data models

[*]Preliminary versions of this article appeared as [18, 19].

[†]University of Limburg, Dept. WNIF, 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: vdbuss@ccu.uia.ac.be

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

are transformed into a conceptual data model such as the relational model [33]. The required database language features then become those of the conceptual model.

Certain semantic and object-oriented data models are equipped with their own data language [8, 21]. DAPLEX [31], for example, is a data language for the Functional Data Model. Even though databases in DAPLEX can be conveniently specified as graphs, queries are formulated textually and can become quite cumbersome.

The first graphical database end-user interfaces were developed for the relational model (e.g., Zloof's Query-By Example (QBE) [37]). The earliest graphical database end-user interfaces for semantic models were associated with the Entity-Relationship Model [15, 30, 35]. Subsequently, graphical interfaces were developed for more complex semantic object-oriented database models [9, 17, 24, 25, 27]. Graph-oriented end-user interfaces have also been developed for recursive data objects and queries [13, 20, 37]. Unfortunately, most interfaces using graphs as their central tool are rather limited in expressive power, as far as data languages are concerned.

Most object-oriented database models, on the other hand, offer computationally complete data languages. However, these languages are mostly non-graphical, usually in the style of object-oriented programming languages such as Smalltalk [16]. Due to their expressiveness, these languages do not lend themselves easily as high-level data languages [6, 36].

It is our purpose to introduce a graph-based object-oriented data model, called the *Graph-Oriented Object Database Model (GOOD)*. In *GOOD*, both the representation and manipulation of data are done graphically, in a very uniform manner. In this respect, *GOOD* can fully take advantage of graphical user interfaces. Furthermore, *GOOD*'s graphical query language can be shown to be computationally complete "up to copy-elimination" (cfr. [3]), thus satisfying the expressiveness conditions usually imposed on object-oriented data manipulation languages.

This article is further organized as follows. In Section 2 we define how graphs represent the object database schemes and instances. Section 3 introduces the five primitive operations: node addition, edge addition, node deletion, edge deletion and abstraction, and a method construct. In Section 4 a number of powerful macros are defined. Finally in Section 5, we focus on object-oriented concepts.

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 [11] storing documents which may contain text, graphics or sound information. The (directed) graph in Figure 1 shows a possible object base scheme for such a system. The oval (rectangular) nodes in this graph represent printable (non-printable) objects and are labeled with printable (non-printable) object labels. For example, the oval shaped node with label *String* represents a character string. The rectangular-shaped node with label *Info-Node* represents a node of information in the hyper-media system.

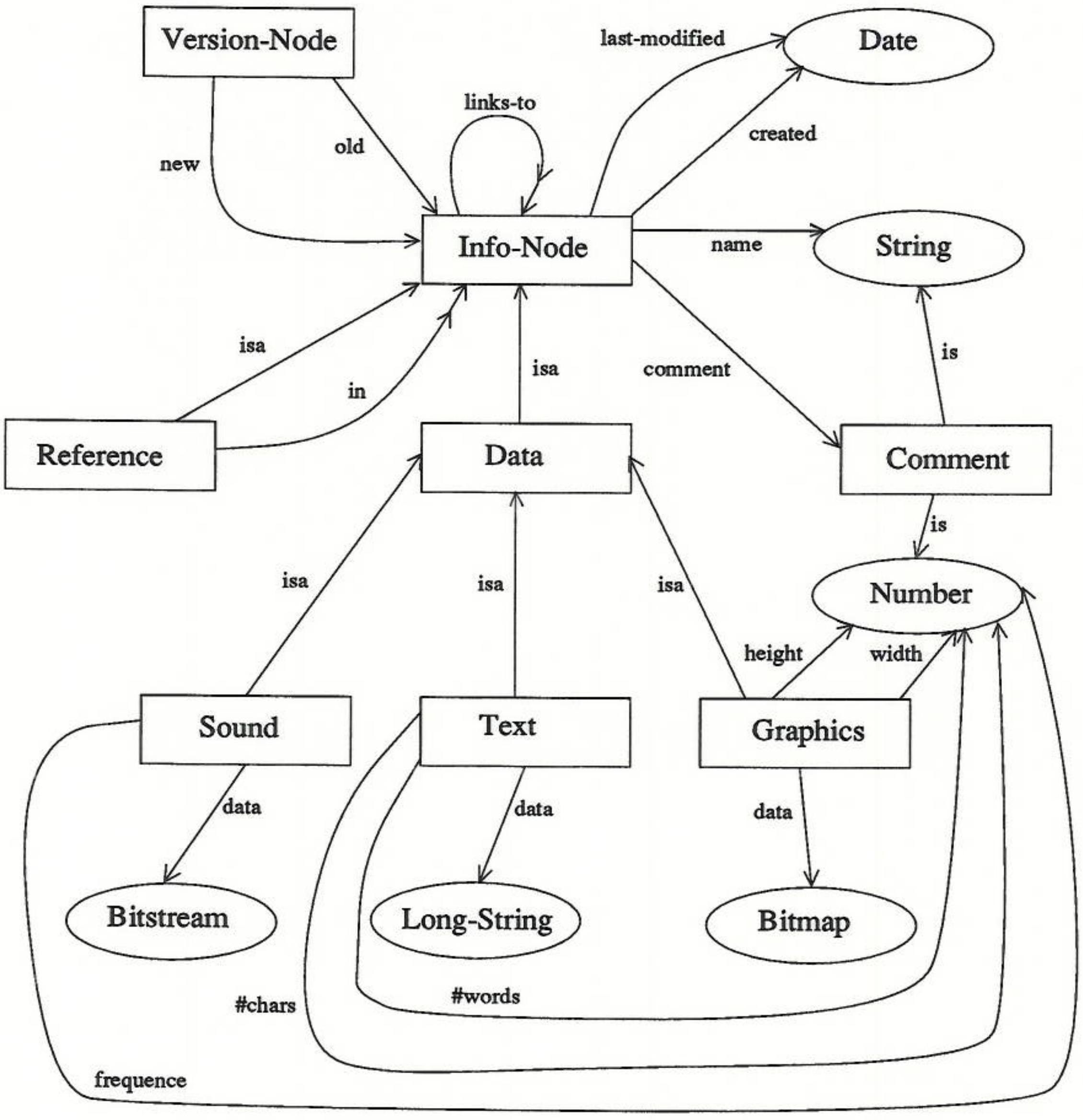


Figure 1: The hyper-media object base scheme.

The single (double) arrowed edges represent relationships and are labeled with functional (non-functional) edge labels. A functional edge represents a *unique* relationship between the two nodes it connects. For example the functional edge *created* indicates that each info-node has a unique creation-date. A non-functional (also called multivalued) edge also represents a relationship between the nodes it connects. However, there can be multiple non-functional edges leaving the same node. So a given info-node can be linked to multiple other info-nodes.

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 non-functional.

Associated with a graphics node are its 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 are now ready for the formal definition of an object base scheme. Throughout this article, we assume there are infinitely enumerable sets of *nodes*, *non-printable object labels*, *printable object labels*, *functional edge labels* and *non-functional edge labels*. These four sets of labels are assumed to be pairwise disjoint. We also assume there is a function π which associates to each printable object label a set of *constants* (e.g., characters, strings, numbers, booleans, ..., but also drawings, graphics, sound, etc).

An object base scheme *is a five-tuple* $\mathcal{S} = (NPOL, POL, FEL, NFEL, \mathcal{P})$ *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*
- $\mathcal{P} \subset NPOL \times (NFEL \cup FEL) \times (NPOL \cup POL)$.

An object base scheme is represented by a directed graph with two kinds of nodes: oval-shaped, labeled by a label of POL, and rectangular-shaped, labeled by a label of NPOL, and two kinds of edges, functional edges (shown as "$\rightarrow$"), labeled by labels of

FEL, and non-functional edges (shown as "$\rightarrow\!\!\!\rightarrow$"), labeled by labels of NFEL. For every triple in $\mathcal{P}$ we have an edge in 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 satisfying the constraints specified in the scheme.

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, 1991* is repeated seven times. In reality, only one such node appears in the object base instance, obviously with six 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 non-functional edges labeled with *linked-to*) to three other info-nodes, i.e., 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.

As a final remark, notice how the graph representing the hyper-media object base instance is connected. In general, however, an object base instance over the hyper-media scheme does *not* have to be represented by a connected graph.

We are now ready to formally define object base instances.

Let $\mathcal{S} = (NPOL, POL, FEL, NFEL, \mathcal{P})$ 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*

[1]We should note that we do not intend to present this typically large and complex graph as such to the user. To display its contents, more organized representations, such as tables, can be used.

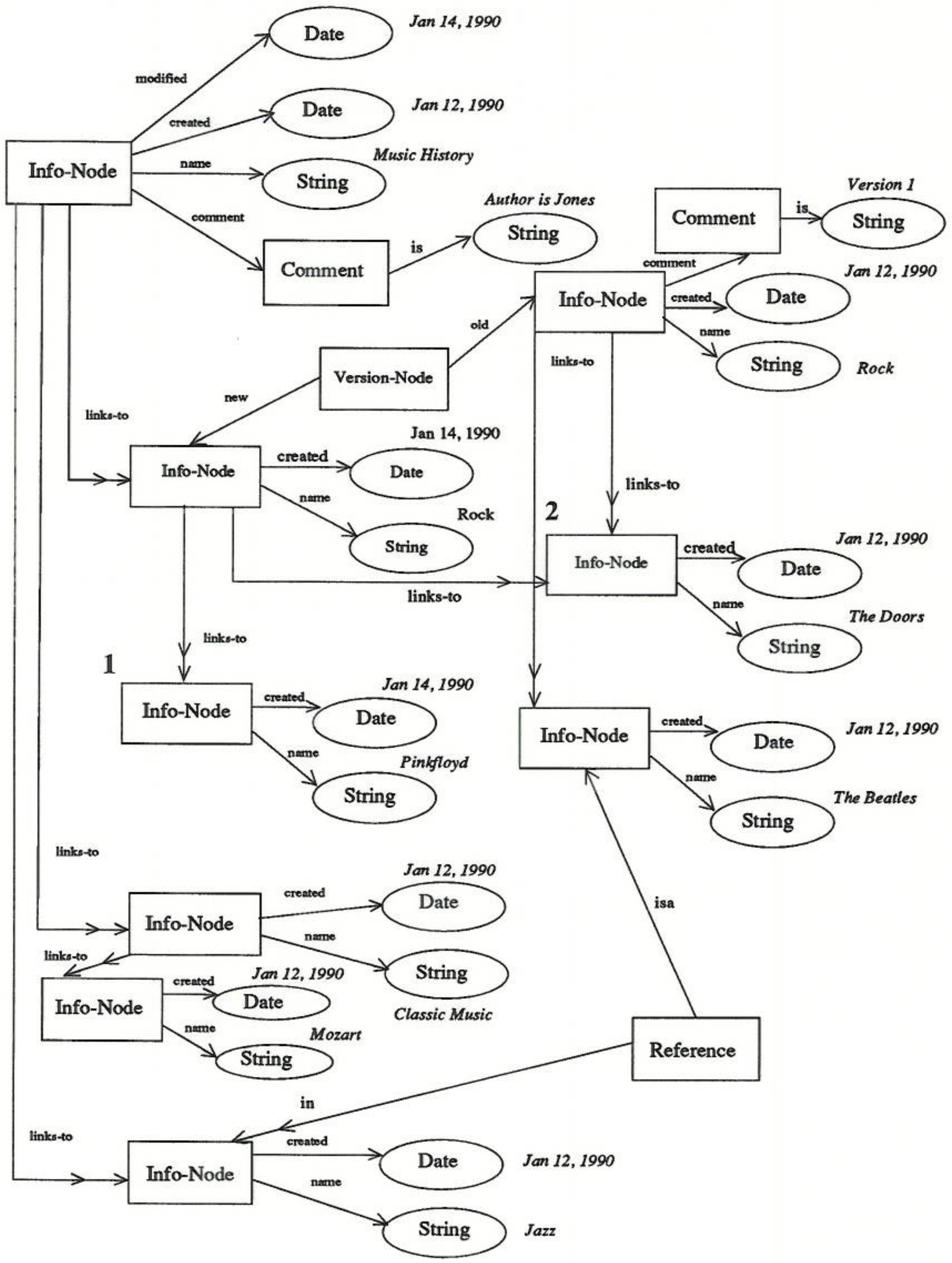


Figure 2: An example of a hyper-media object base instance.

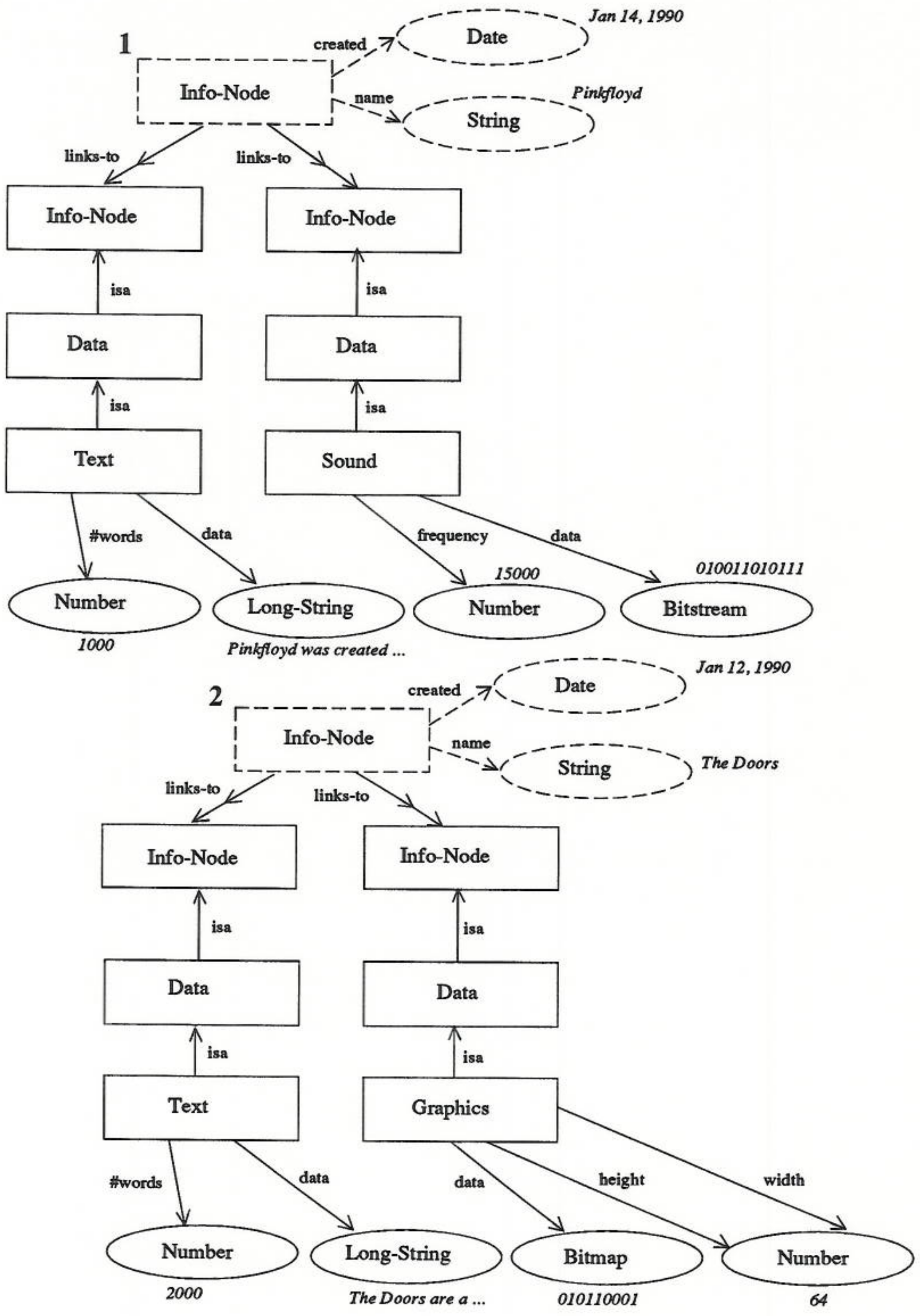


Figure 3: Continuation of the hyper-media object base instance.

be in NPOL $\cup$ POL; if $\lambda(\mathbf{n})$ is in NPOL (respectively in POL), then $\mathbf{n}$ *is called a* non-printable node *and is represented by a rectangular-shaped node (respectively a* printable node *and is represented by a oval-shaped node);*

- *each printable node* $\mathbf{n}$ *in* $\mathbf{N}$ *has an additional label* $\mathrm{print}(\mathbf{n})$, *called the* print label*;* $\mathrm{print}(\mathbf{n})$ *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 $\cup$ NFEL, and* $(\lambda(\mathbf{m}), \alpha, \lambda(\mathbf{n})) \in \mathcal{P}$*; if* $\lambda(\mathbf{e})$ *is in FEL (respectively in NFEL), then* $\mathbf{e}$ *is called a* functional edge *(respectively a* non-functional 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* $\mathrm{print}(\mathbf{n}_1) = \mathrm{print}(\mathbf{n}_2)$ *then* $\mathbf{n}_1 = \mathbf{n}_2$.

3 The transformation language

The *GOOD* data transformation language is a graphical database language. It contains six graph transformation operators. Four of these correspond to elementary manipulations of graphs: addition of nodes, addition of edges, deletion of nodes and deletion of edges. The fifth operator, called abstraction, is used to group objects on the basis of some of their properties. The sixth operator, called method call, corresponds to method calls in object-oriented database systems. As will be evident from the examples, these operators can be used to query, to update as well as to restructure object bases.

Throughout this article, we will apply the operators to the object base instance of Figure 2 (and continued in Figure 3), unless explicitly specified otherwise.

The specification of all operators relies on the notion of *pattern*. A pattern is a graph used to describe subgraphs in an object base instance. Consider the graph in Figure 4. This graph is a pattern over the hyper-media object base scheme. Intuitively, it describes an info-node, created on *Jan 14, 1991*, with name *Rock* which is linked to another info-node. Formally:

A pattern *over an object base scheme S is a finite object base instance over S, with the exception that a pattern may contain printable nodes without print labels.*

In order to specify the subgraphs in an object base instance corresponding to a pattern, we need to introduce the concept of *embeddings*. The pattern in Figure 4 can be embedded within the instance of Figure 2 in two different ways (see Figure 5 and Figure 6). 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* $\mathcal{S}$. *An* embedding *of* $\mathcal{J}$ *in* $\mathcal{I}$ *is a total mapping* $i\colon \mathbf{M} \to \mathbf{N}$ *preserving all labels, i.e., node labels, edge labels as well as print labels (where specified).*

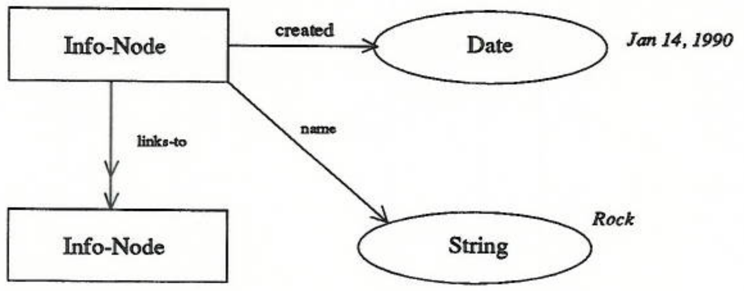


Figure 4: An example of a pattern

We can now start with the discussion of the six operators of the *GOOD* transformation language.

3.1 Node addition

Suppose that we want to identify the info-nodes to which the info-node with name *Rock* and date *Jan 14, 1991* is linked. Therefore, reconsider the pattern in Figure 4. As indicated before, there are two such info-nodes identified by the two different embeddings 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, 1991*. The second node is the info-node with name *Pinkfloyd* and created on *Jan 14, 1991*. A way to identify 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 7. 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 embedding 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 embedding. Figure 8 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 most general form, it can introduce objects that represent aggregates of multiple nodes in the object base instance under consideration. Consider the pattern in Figure 9. It 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 embeddings 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. This can be accomplished by the node addition (Figure 10). The four added nodes will have the node label *pair*, and will be attached with functional edges (labeled *parent* and *child*) to the appropriate creation

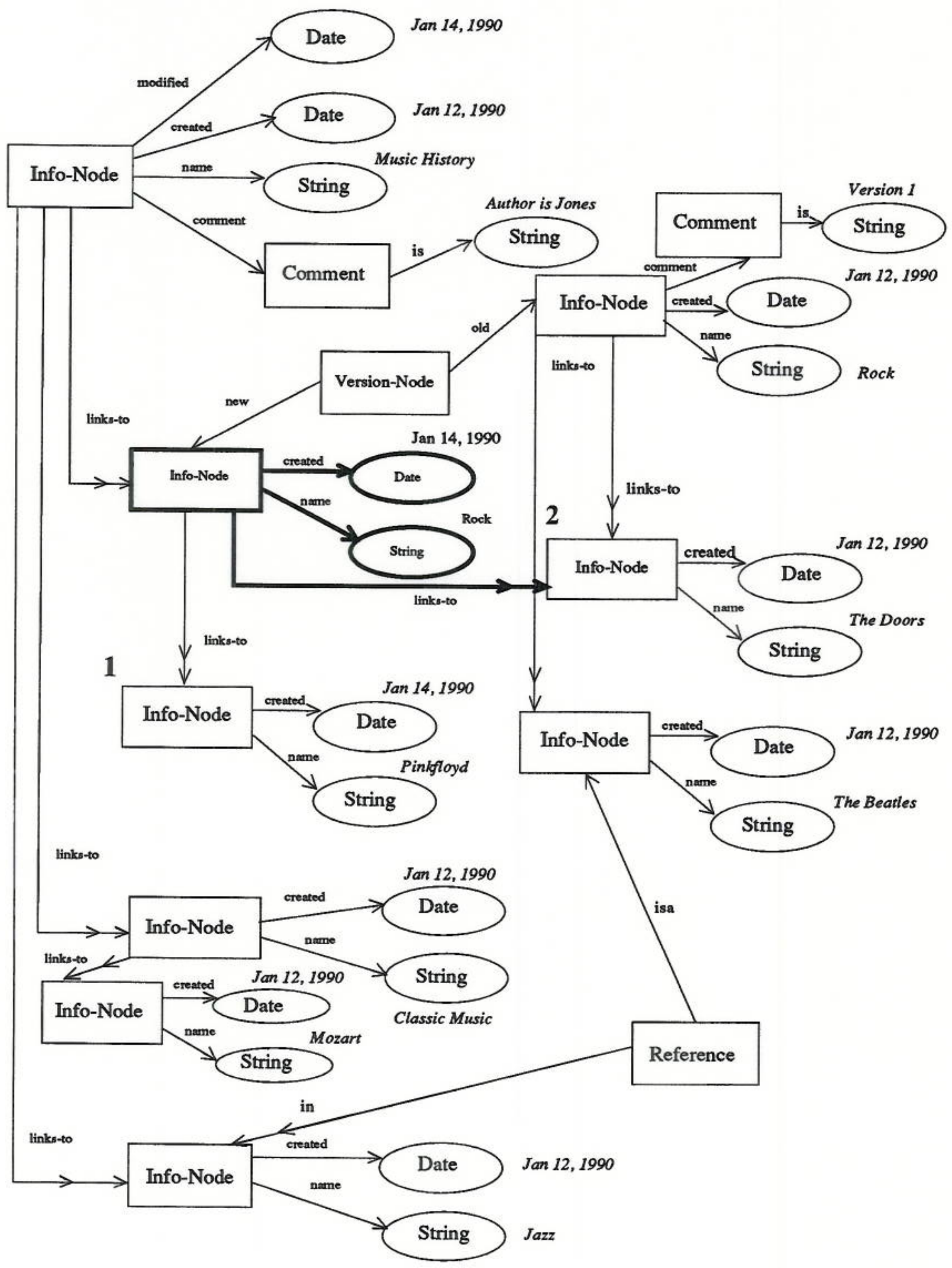


Figure 5: A first way to embed the pattern of Figure 4 within the instance of Figure 2

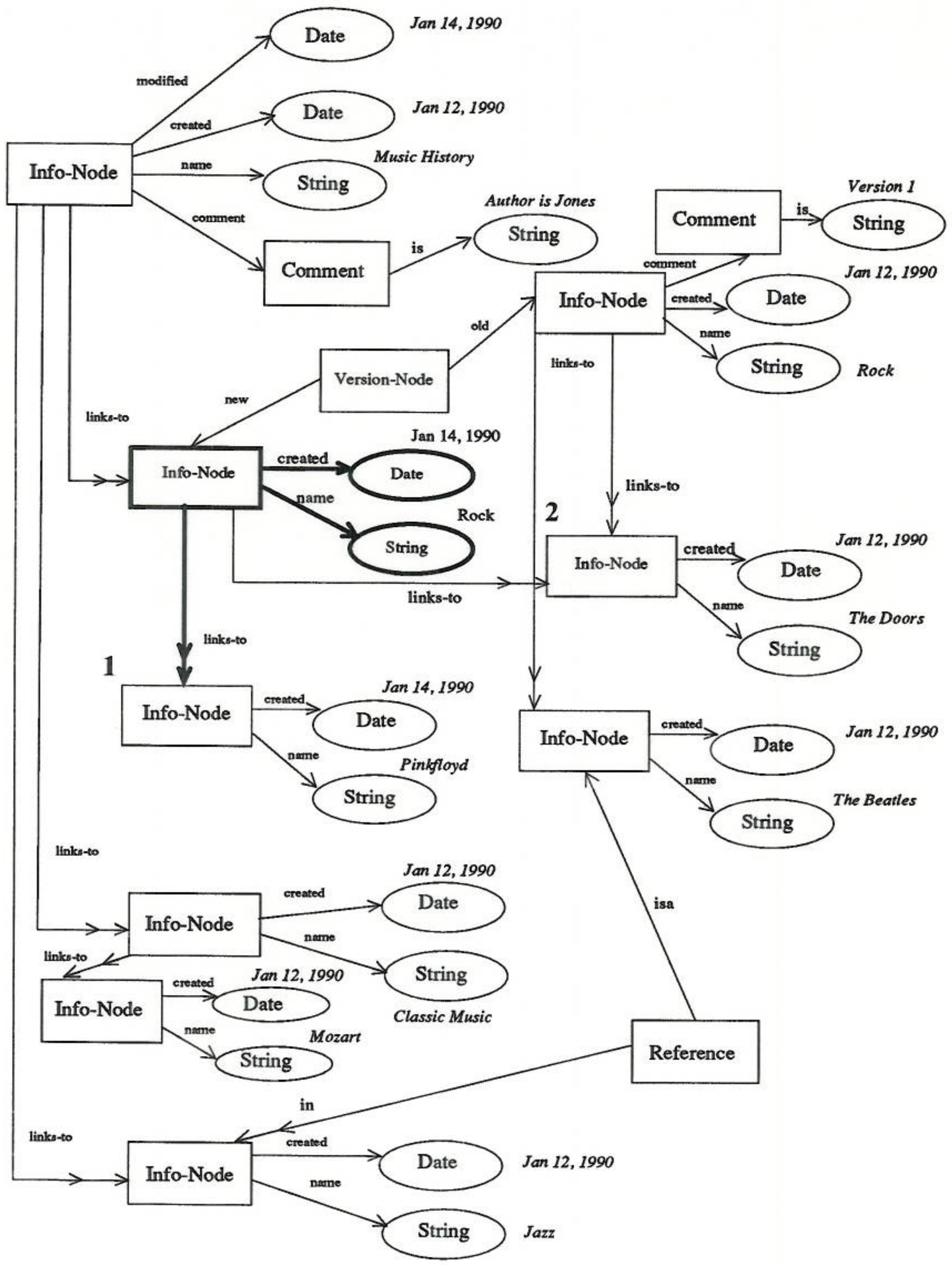


Figure 6: A second way to embed the pattern of Figure 4 within the instance of Figure 2

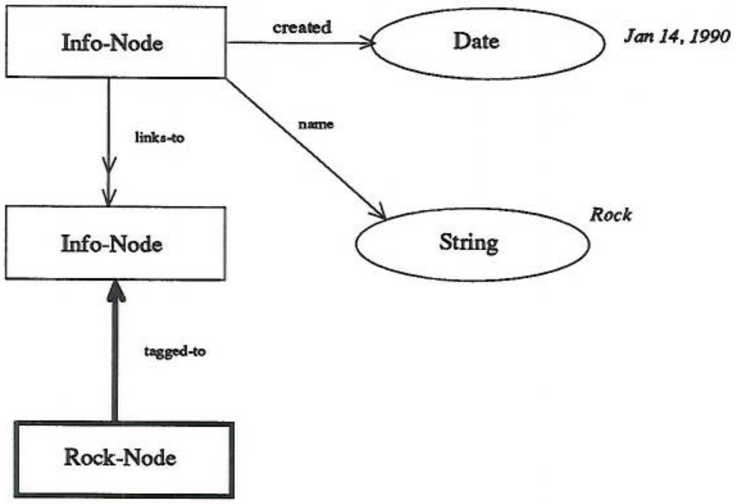


Figure 7: An example of a node addition operation

dates.

As can be seen from the two previous examples, node additions only introduce non-printable nodes. This is based on our assumption that printable nodes exist without having to 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 nodes that are being added and the embeddings that satisfy the source pattern, the reason of course being that there can only be one edge with a specific functional label leaving a newly added node.

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* $\mathbf{m}_1, \dots, \mathbf{m}_n$ *be nodes in* $\mathbf{M}$*. Let* K *be a non-printable object label and let* $\alpha_1, \dots, \alpha_n$ *be functional edge labels.*

The node addition

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

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* $(\mathbf{m}, \alpha_1, \mathbf{m}_1), \dots, (\mathbf{m}, \alpha_n, \mathbf{m}_n)$*;*
- $\mathcal{S}'$ *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*

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

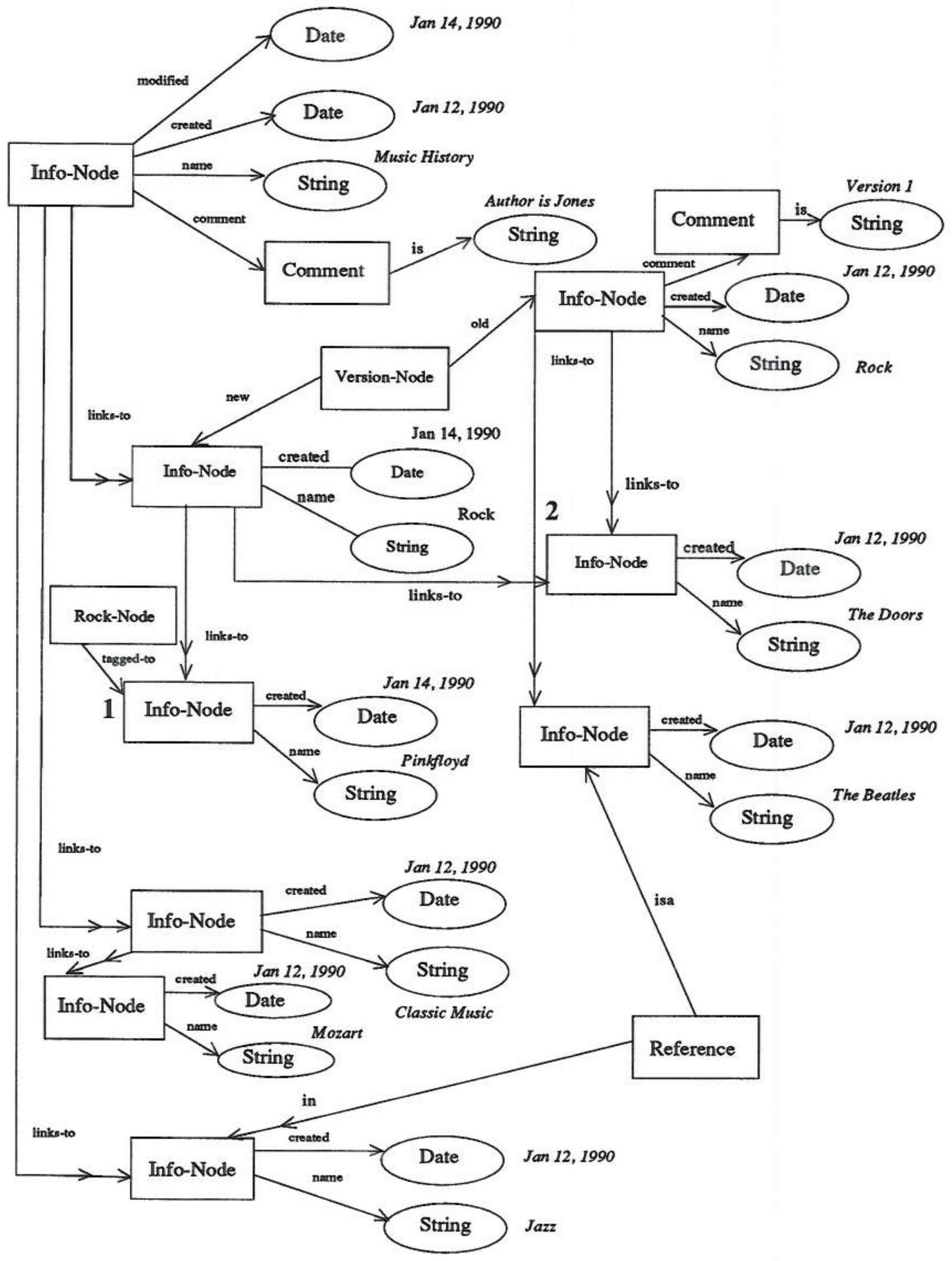


Figure 8: The result of a node addition

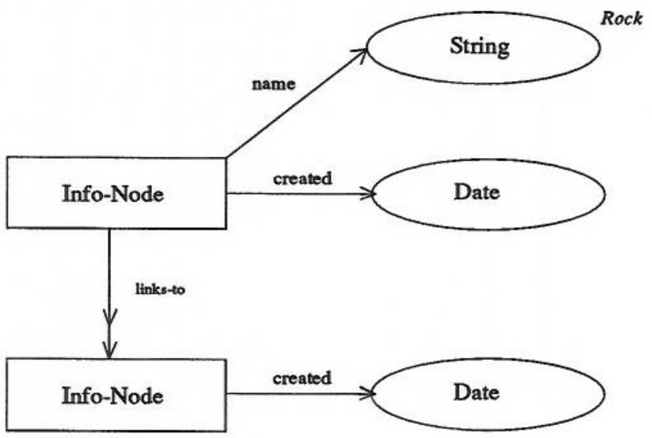


Figure 9: Another example of a pattern over the hyper-media object base

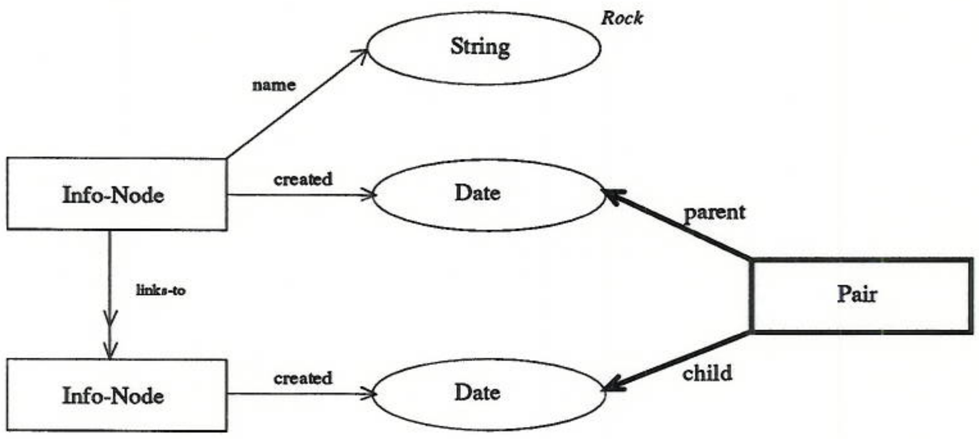


Figure 10: The specification of a node addition

```
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 non-printable 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 embedding $i$ 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 11: Procedural semantics of node addition.

1. *$\mathcal{I}$ is a subinstance of $\mathcal{I}'$;*
2. *for each embedding 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*
3. *each edge in $\mathcal{I}'$ leaving a node of $\mathcal{I}$ is also an edge of $\mathcal{I}$.*

Node addition is always well-defined if the α_i are all distinct.

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 11. The algorithms for the other operations are similar.

3.2 Edge addition

The node addition operator can be used to introduce new objects into an object base. The edge addition operator, 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, 1991*. 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 12. 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. If we assume that a document has a unique creation date, we may assume that this edge is functional. Figure 13 shows that part of the hyper-media object base affected by this edge addition.

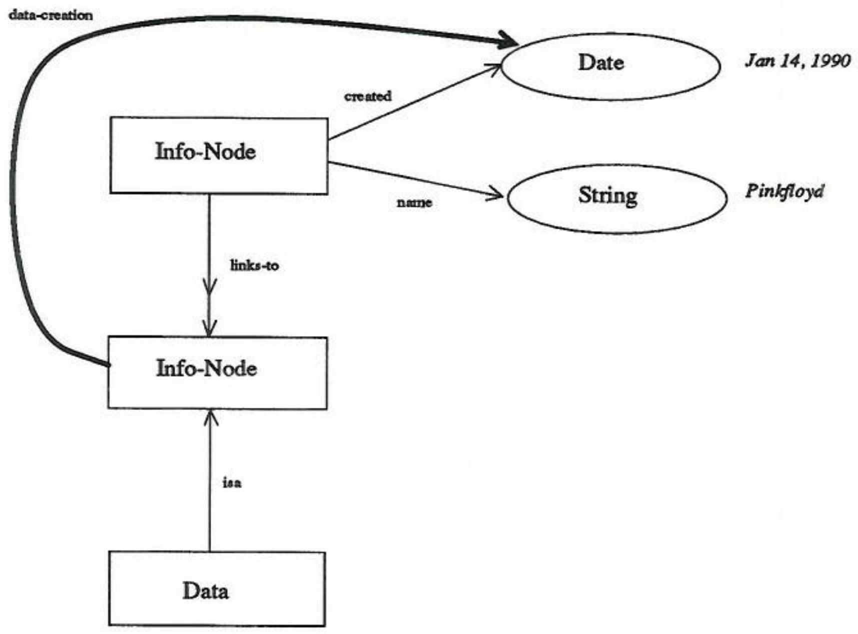


Figure 12: An example of an edge addition

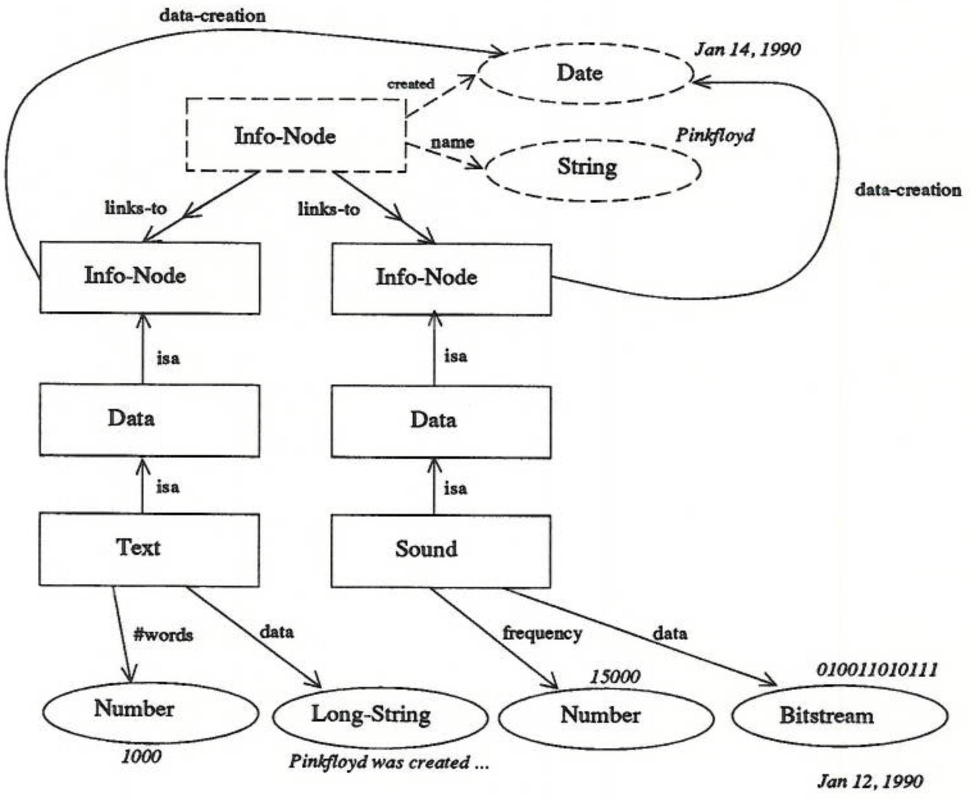


Figure 13: The result of an edge addition

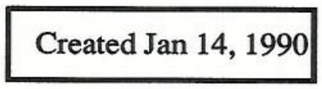


Figure 14: Adding a single node labeled *Created on Jan 14, 1991*

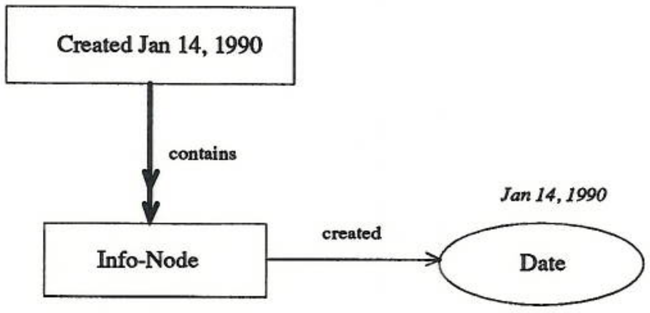


Figure 15: Linking to *Created on Jan 14, 1991* the info-nodes created on Jan 14, 1991

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, 1991*. 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 14 (notice 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 non-printable node with label *Created on Jan 14, 1991*. The second step consists of connecting to this newly created node all the info-nodes created on *Jan 14, 1991.* This is accomplished with the edge-addition shown in Figure 15. Notice that this edge addition introduces non-functional edges. This is necessary since in general there is more than one info-node created on the indicated date.

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;*

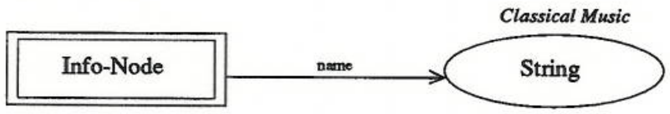


Figure 16: Example of a node deletion

- *$\mathcal{I}'$ is the minimal instance over S' for which $\mathcal{I}$ is a subinstance of $\mathcal{I}'$, 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 $\mathcal{I}'$.*

Observe that the result of an edge addition is not defined if the addition of the required edges would yield different edges with (i) 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* operators, statically checking the "consistency" of an edge addition in the program is undecidable in general, as can be shown using results from [2, 26]. So in general, some limited run-time checks have to be 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 operator.

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 16. Again this figure has two distinguishable parts. The first part is the source pattern which as always determines the relevant embeddings. The second part consists of a single node (in double outline) specifying the nodes to be deleted. Figure 17 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 operator 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, shown in Figure 18, attaches to each data info-node a node with label *No Sound*. The second step, shown in Figure 19, removes the tag nodes of data info-nodes containing sound data. The remaining tagged info-nodes are those that do not contain sound data. Figure 20 shows the relevant part of the hyper-media object base obtained by this process.

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

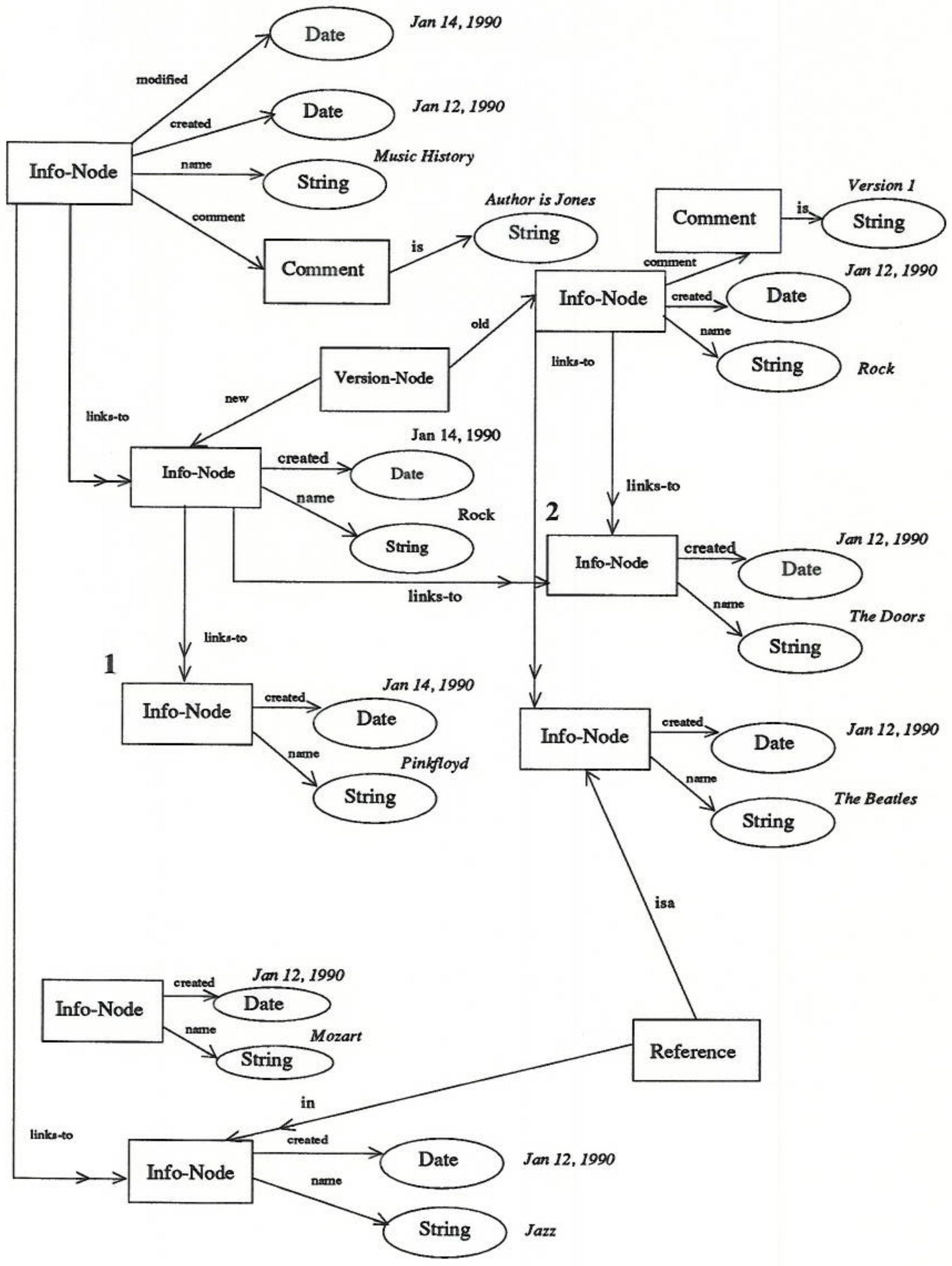


Figure 17: Result of a node deletion

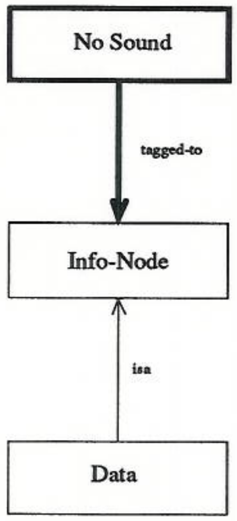


Figure 18: Tagging data info nodes

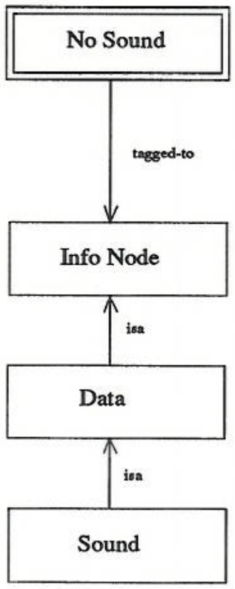


Figure 19: Removing tags of info-nodes containing graphics nodes

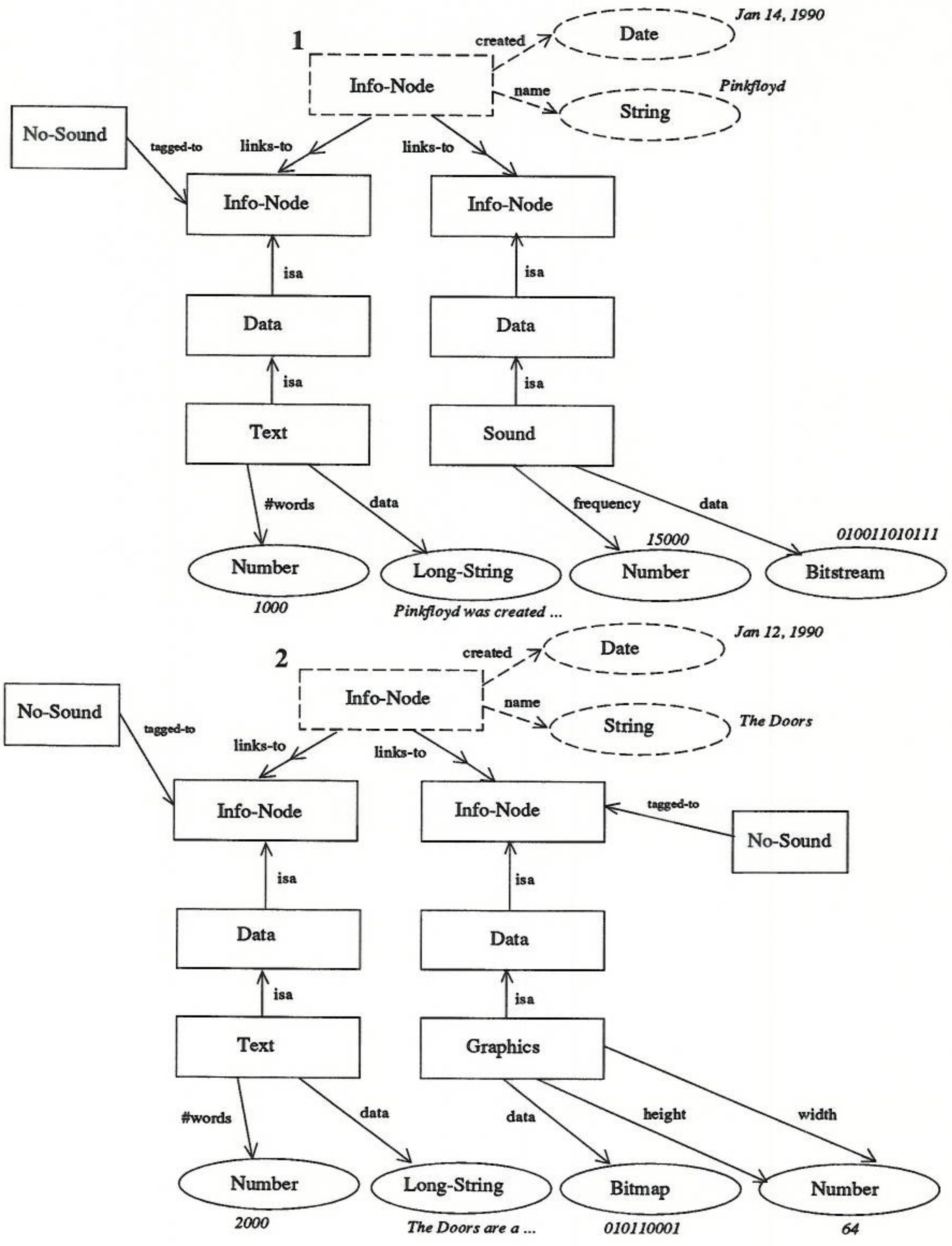


Figure 20: The result of a node addition and a node deletion

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 non-printable 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:

- *$\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ 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 $\mathcal{S}'$ for which $\mathcal{I}'$ is a subinstance of $\mathcal{I}$, and such that for each embedding 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 operator.

Suppose we modified the info-node with name *Music History* on Jan 16, 1991. Consequently, we need to update the *last-modified* property from *Jan 14, 1991* to *Jan 16, 1991*. This can be done in two steps. The first step, shown in Figure 21, 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 21, adds a new edge resulting in the intended update. The result of these two operations is shown in Figure 22. Although this is not illustrated in this example, it should be clear that it is also possible to remove non-functional 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 $\mathcal{F}$.*

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*

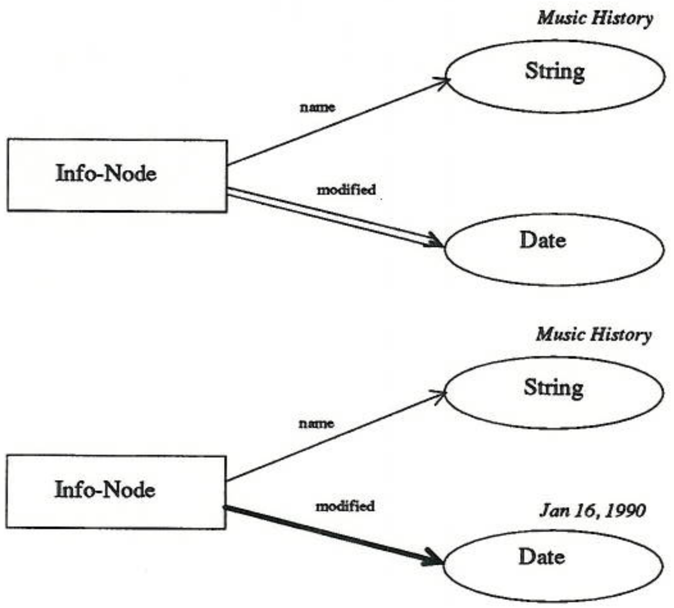


Figure 21: Example of an update with an edge deletion followed by an edge addition

- *$\mathcal{I}'$ is the maximal instance over S' for which $\mathcal{I}'$ is a subinstance of $\mathcal{I}$, 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 not labeled edges of $\mathcal{I}'$.*

3.5 Abstraction

GOOD is designed to support object identity. This means that objects can be distinguished independent of their associated values or properties. Sometimes, however, it is desirable to "abstract" over objects that share the same values or properties. The operator supporting this technique in *GOOD* is the abstraction operator. The abstraction operator creates objects that are *defined* by those values or properties under consideration.

Reconsider the hyper-media scheme specified in Figure 1. This scheme allows for the maintenance of different versions of info-nodes. Now consider Figure 23. This figure displays a sub-instance of an hyper-media object base instance different from the one displayed in Figure 2 and Figure 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 third info-nodes from the left. In order to "abstract" over info-nodes which share the same set of nodes, consider Figure 24. 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

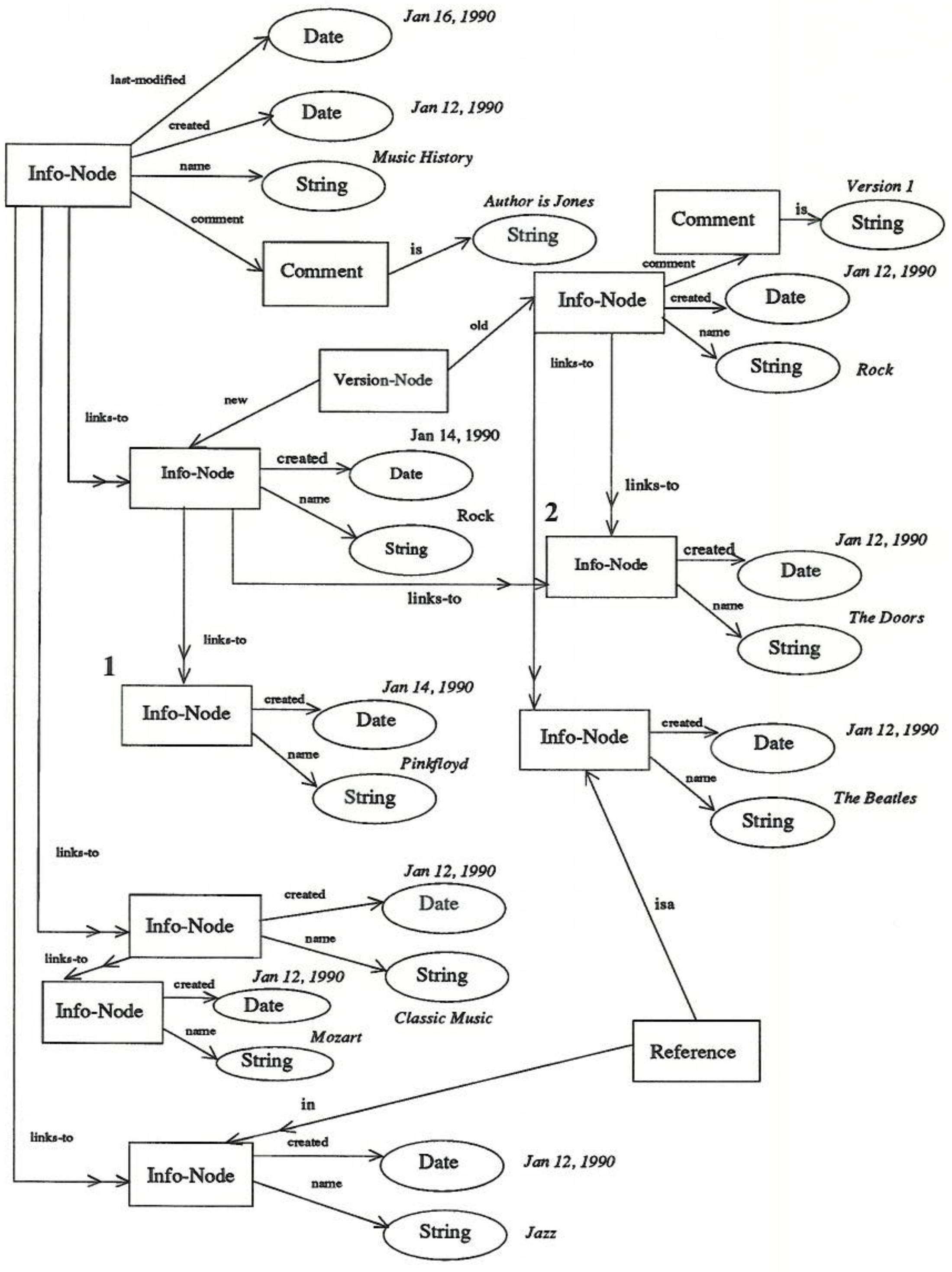


Figure 22: The result of an edge deletion followed by edge addition

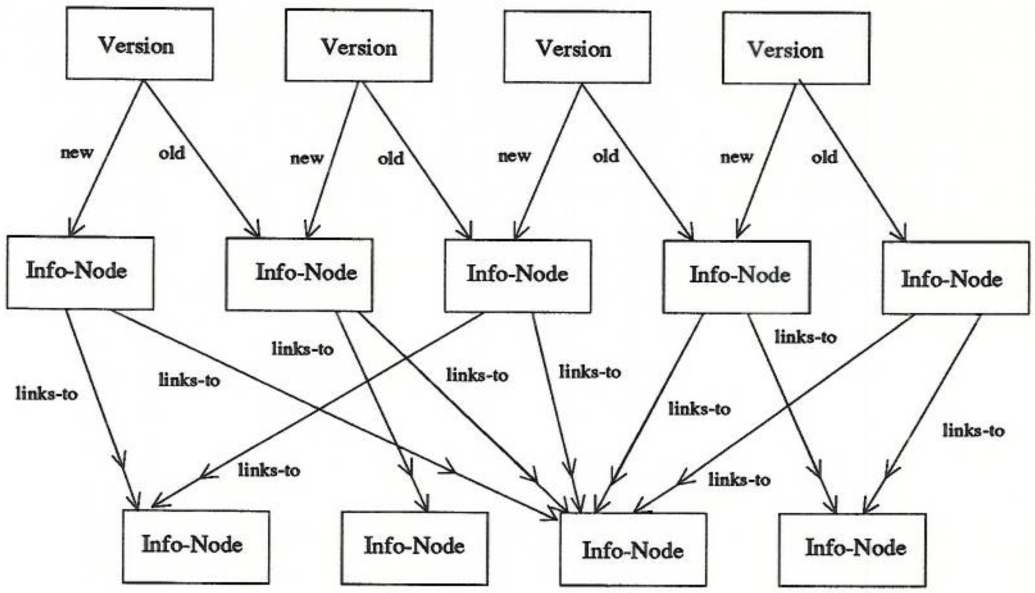


Figure 23: A sequence of versions of related information

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 25 (for clarity, we have omitted the T and *interested* nodes).

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 non-printable node in $\mathbf{M}$. Let K be a non-printable object label, and let α, β be non-functional 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

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

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 non-functional 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 $\mathcal{S}'$ for which*

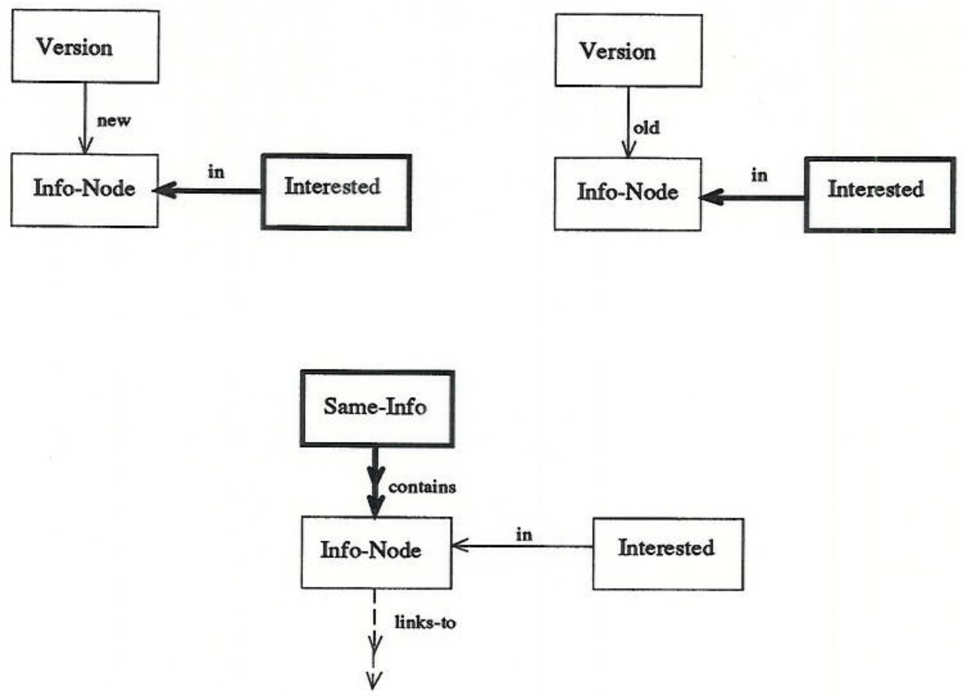


Figure 24: Example of an abstraction operation

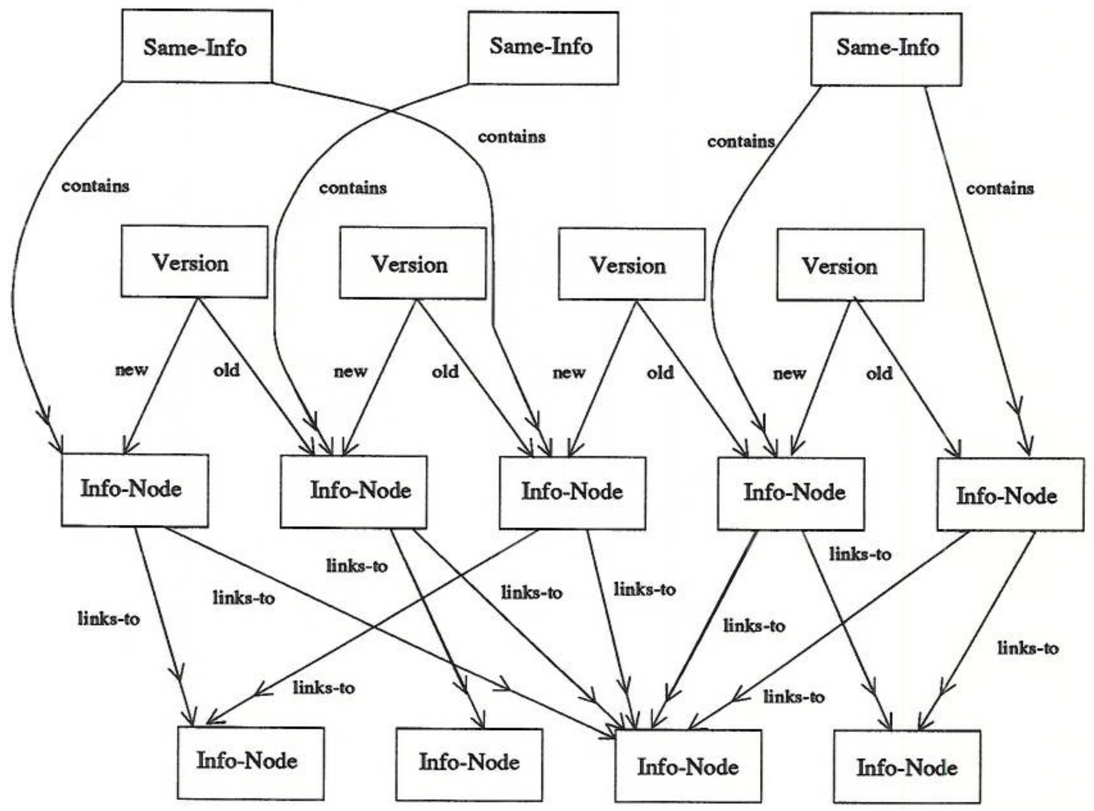


Figure 25: Instance after an abstraction operation

1. *$\mathcal{I}$ is a subinstance of $\mathcal{I}'$;*
2. *for each embedding i of $\mathcal{J}$ in $\mathcal{I}$, there exists a K-labeled node $\mathbf{p}$ in $\mathcal{I}'$ such that $(\mathbf{p}, \beta, i(\mathbf{n}))$ is a non-functional 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 node $\mathbf{r}$ in $\mathcal{I}'$, $(\mathbf{q}_1, \alpha, \mathbf{r}) \in \mathbf{E}' \Leftrightarrow (\mathbf{q}_2, \alpha, \mathbf{r}) \in \mathbf{E}'$; and*
4. *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. 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* operators 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 [14].

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*.

Reconsider the node addition specified in Figure 18 and the node deletion specified in Figure 19. This sequence of operations identifies the data info-nodes without sound data nodes. We can specify this sequence within a single method. Consider Figure 26. The top part of this figure is a method specification. It contains, within a regular diamond-shaped node, the name the method (in this case *Soundless*). This node has an edge (without a label) that points to the class of the receiving objects (in this case the info-nodes). If the method has additional parameters, they are also indicated (in this case there are no parameters). The bottom part of Figure 26 contains the sequence of operations to be performed by the method (in this case the above mentioned node addition and node deletion). These operations are specified exactly as before, except for the addition of diamond shaped nodes. These diamond shaped nodes serve to identify the receiving object of the corresponding method call (they are essentially the analogues of the *self* variables in object-oriented programming languages).

Now consider Figure 27, which represents a method call. Intuitively, the *Soundless* method is applied to all info-nodes. The result of this method call is exactly the same as if the two operations in the method body were applied separately. Notice how the diamond-shaped node and the arrow pointing to the receiving object are drawn in bold.

The main reason for this notation stems from the fact that method calls can be invoked within method bodies. The bold notation then serves to distinguish between the receiving objects (pointed at by regular diamonds) and method calls (bold diamonds).

As is the case with all previously introduced *GOOD* operations, a method call can be invoked in the context of a source pattern. For example, suppose we want to call the method *Soundless* only for those info-nodes that were created on January 14, 1991. This can be accomplished with the method call in Figure 28.

As previously mentioned, methods may involve parameters. Consider specifying a method to update the last modification date of an info-node. A natural parameter to this method is the proposed modification date. The method specification and method body shown in Figure 29 fulfill this task: the method specification identifies as receivers info-nodes and as parameter a date. The method body specifies that the current *last-modification* property of the receiver should be dropped (using an edge deletion) and replaced by the date supplied by the parameter (using an edge addition). The method call shown in Figure 30 changes the last modification date of the info-node with name *Music History* to *Jan 16, 1991*.

Methods can also be used to specify recursive processes. Suppose that we want to remove all the old versions of an info-node. Since we do not a priori know the number of old versions of a specific info-node, it is impossible to perform this task by only using the first five operations of the *GOOD* transformation language. We next show how methods can be used to overcome this problem. Consider the method in Figure 31. The method specification introduces the method *Remove Old Versions* with as receivers info-nodes. The method body consist 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.

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

A GOOD method is a named *procedure. It has* parameters, *a* method specification, *and a* method body. *Let S be an object base scheme.*

The method specification contains the method's name and parameters. 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, $s_{\mathcal{M}}: L_{\mathcal{M}} \rightarrow NPOL \cup POL$, with $L_{\mathcal{M}}$ a finite (possibly empty) set of labels in FEL. $s_{\mathcal{M}}$ associates with each of its labels a parameter. $R_{\mathcal{M}} \in NPOL$ 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 $R_{\mathcal{M}}$. No two edges point to a same node.*

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,*

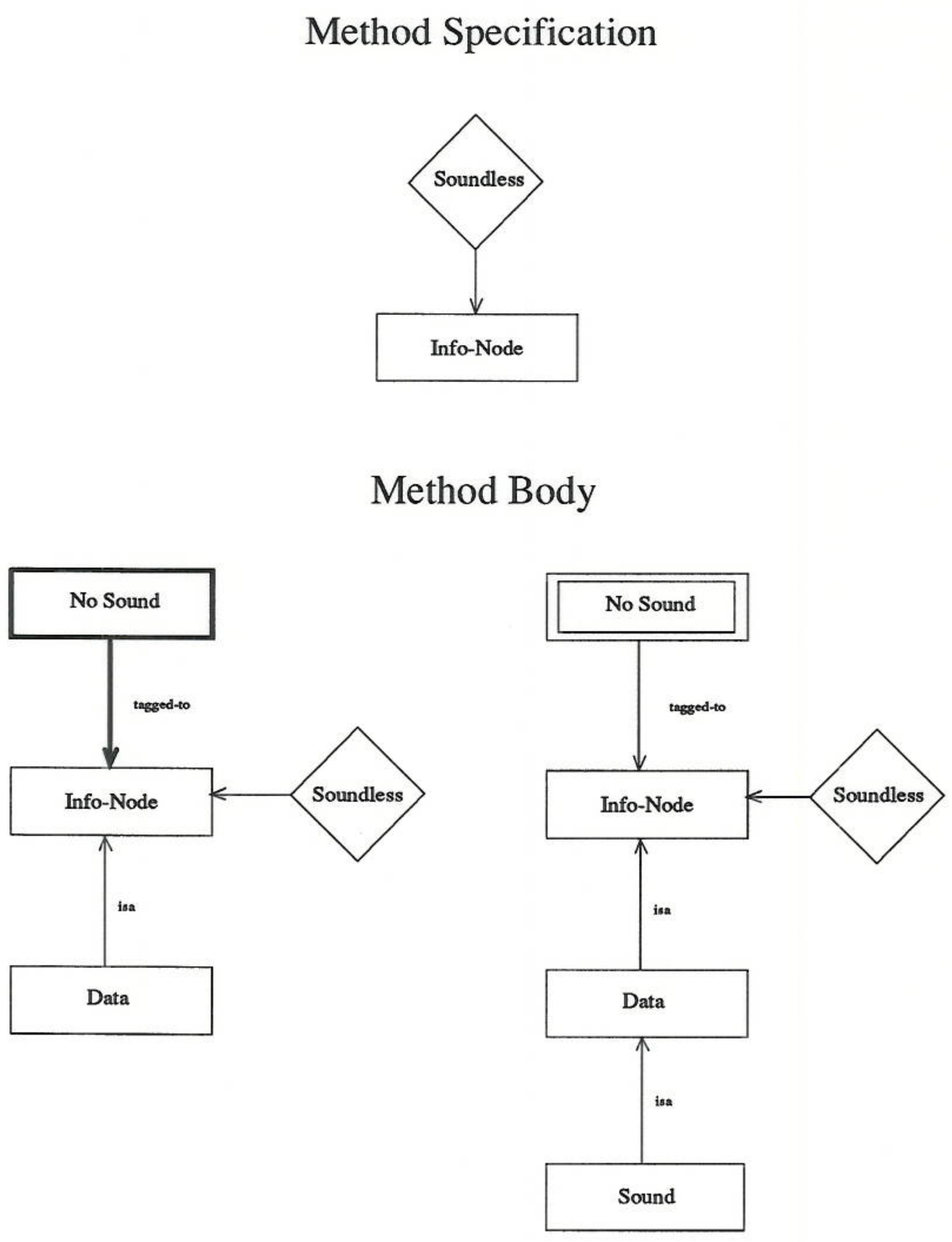


Figure 26: An example of a method

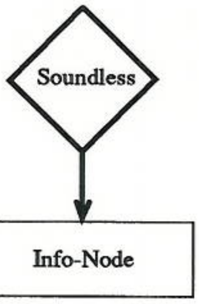


Figure 27: An example of a method call

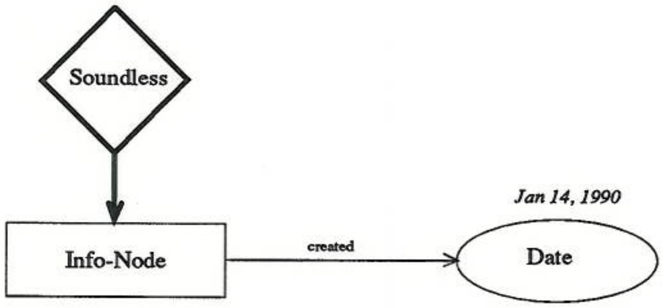


Figure 28: An example of a method call in the context of a source pattern

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 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, $g\colon L_{\mathcal{M}} \to \mathbf{N}'$ 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 these 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 $\mathrm{NA}[\mathcal{J}, \mathcal{S}, \mathcal{I}, K, \{(\beta, g(\beta)) | \beta \in L_M\} \cup \{(\beta_{\mathbf{n}}, \mathbf{n})\}] = (\mathcal{J}_0, \mathcal{S}_0, \mathcal{I}_0)$. *Let*

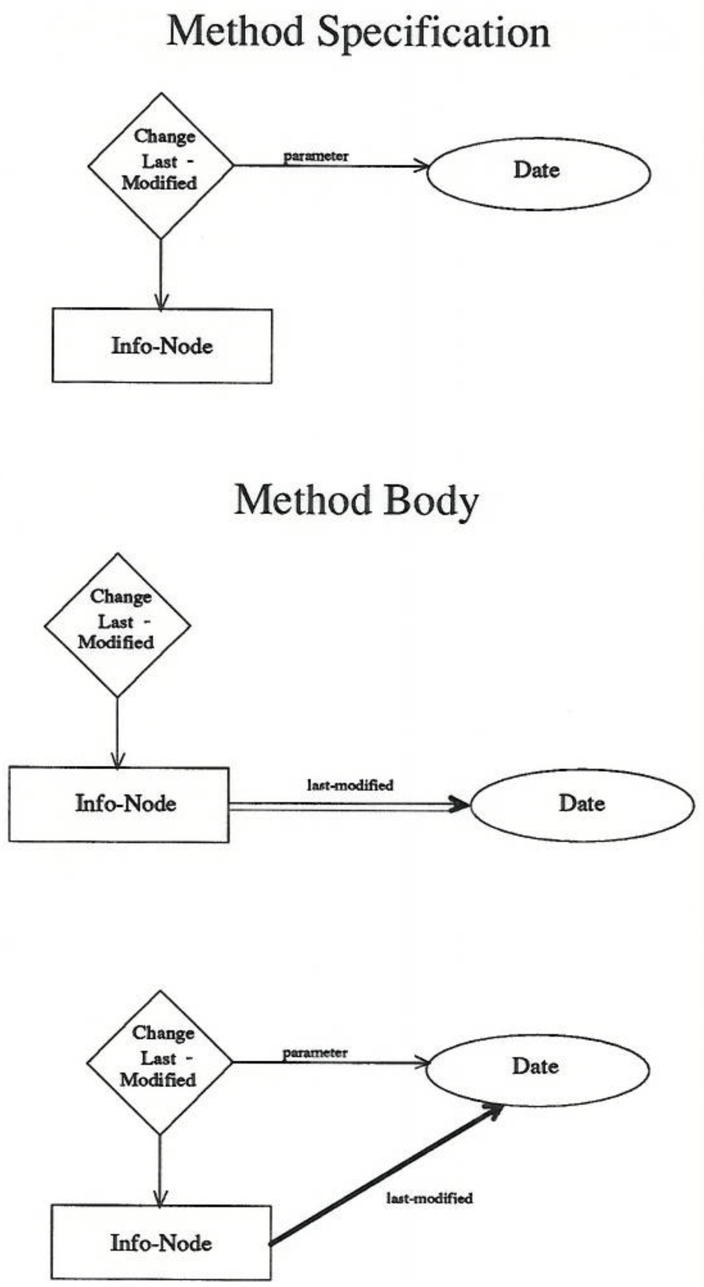


Figure 29: A method to change the last modification date of an info-node

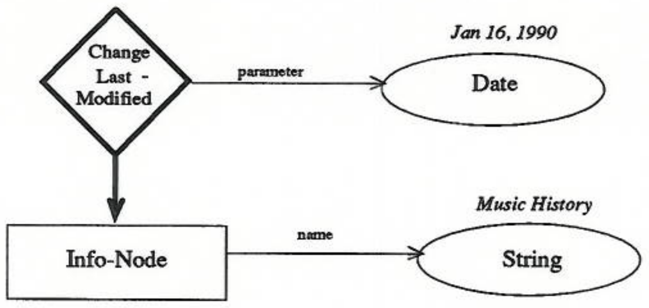


Figure 30: A method call to change the last modification date of the *Music History* info-node

$(PO_1, PO_2, \ldots, PO_k)$ 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 the diamond-shaped node of the source pattern of PO_i is substituted by a rectangular-shaped node labeled by K.*

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 $\mathbf{m}$ labeled K. Finally $\mathcal{S}'$ is defined as $\mathcal{S}_{k+1}$ in which all the triples with first component K are deleted and $\mathcal{I}'$ is defined as $\mathcal{I}_{k+1}$.

3.7 Interpretation modes of the transformation language

The *GOOD* transformation language describes transformations of database graphs. In an actual implementation, these transformations can be interpreted in different ways: as queries, as updates, as scheme manipulations, or as restructurings. We associate to each execution of a *GOOD* program one such interpretation mode. The semantics of a program depends on the associated interpretation mode.

In query mode, the execution of the program does not affect the stored instance or scheme of the database. The program only describes the result of the query. In update mode, the execution of the program affects the instance but not the scheme of the database. The resulting instance is described by the program, while the scheme remains unchanged. In scheme manipulation mode, the database instance only changes through complete deletions of a whole class or property. Finally, in restructuring mode, the scheme and the instance of the database fully transform.

A complete description of these interpretation modes is given in [4].

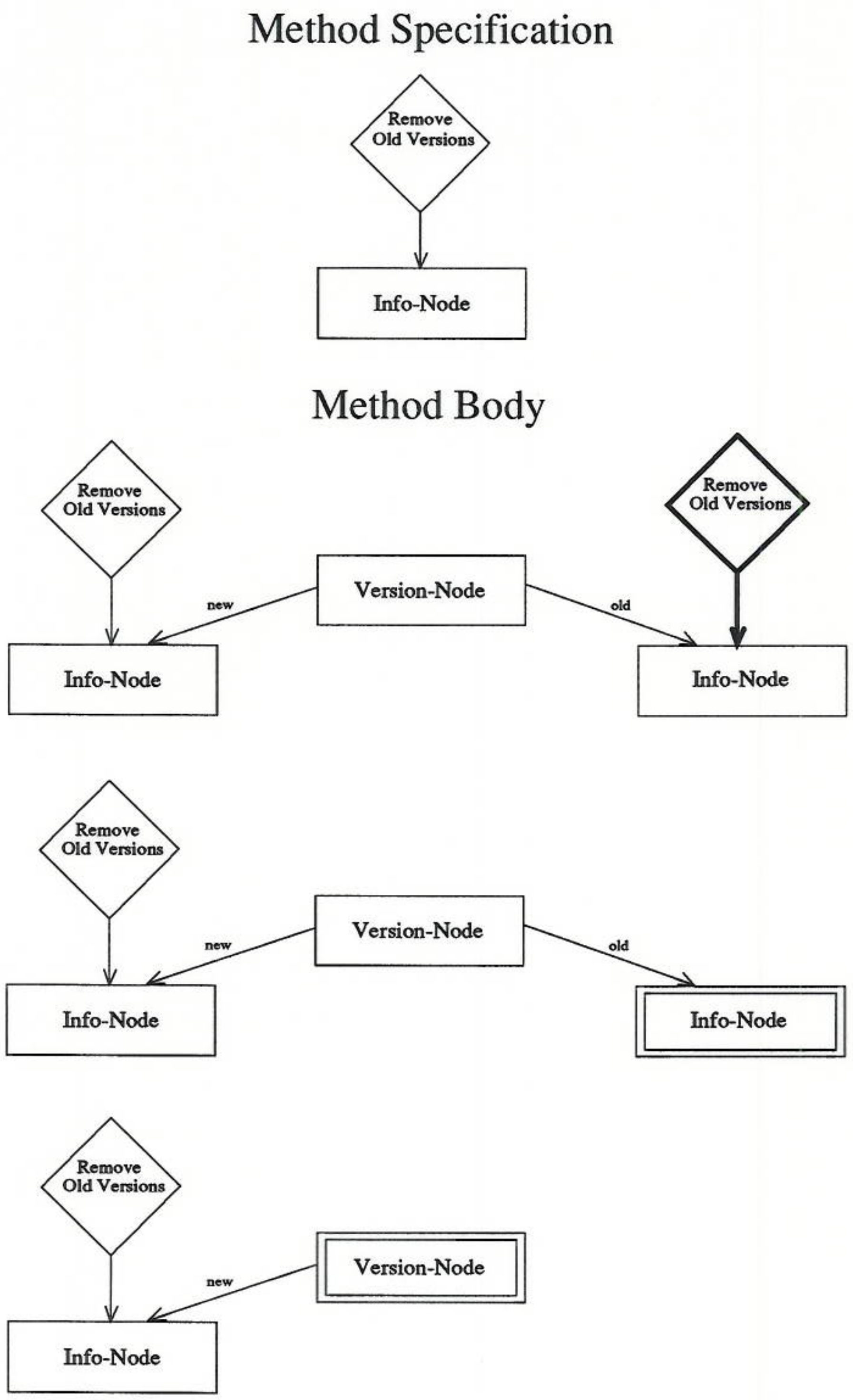


Figure 31: An example of a recursive method to delete old versions of an info-node

4 Macro operations

This section will introduce a variety of additional graphical operations for the *GOOD* transformation language. These operations make data manipulation more succinct, as in [12, 5]. However, they do not increase the expressive power of the *GOOD* transformation language. This is demonstrated by explaining for each macro operation how it can be expressed using the standard operations of Section 3.

4.1 Generalized addition and deletion operations

The *generalized addition* is syntactically as well as semantically a generalization of the node and the edge addition. Its graphical appearance is as follows. Besides the usual pattern, bold edges correspond to the underlying edge addition. Furthermore, several bold nodes may appear, which correspond to the underlying node additions. Note that bold multivalued edges leaving bold nodes are allowed now: they correspond to the underlying edge addition. The effect of generalized addition is as follows: first, the underlying node additions are executed in parallel; then, the underlying edge addition is performed. The general technique to simulate a generalized addition in *GOOD*, is to create an intermediate node for every embedding of the pattern using an auxiliary node addition operation with functional edges to each node of the pattern. The intermediate nodes are then used for each of the node additions. The augmented pattern thus obtained is then used for the edge addition. Finally the intermediate nodes are deleted.

The *generalized deletion* generalizes the node and the edge deletion. It has the same form as these, but any number of edges and any number of nodes can be drawn in double outline. The effect is obvious. The general technique to simulate a generalized deletion in *GOOD* is similar to that of simulating generalized addition: intermediate nodes are created for every embedding of the pattern, and are then used to perform the various deletions consecutively.

The generalized operations are quite useful in practical situations. An example of generalized addition is given later in this section.

4.2 Negation

The pattern embedding 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, a generalized addition, is shown in Figure 32. 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 33 simulates the query of Figure 32. First, intermediate nodes are created for every embedding of the non-crossed part of the pattern. Then the intermediate nodes

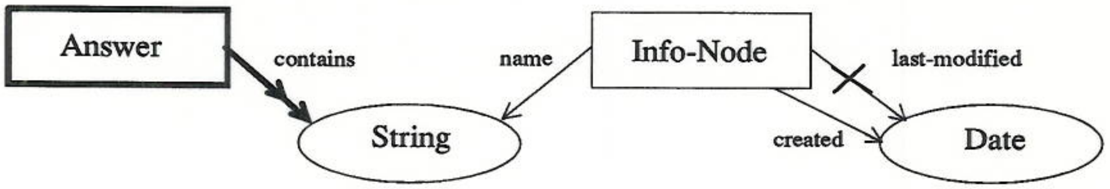


Figure 32: Expressing absence of edges.

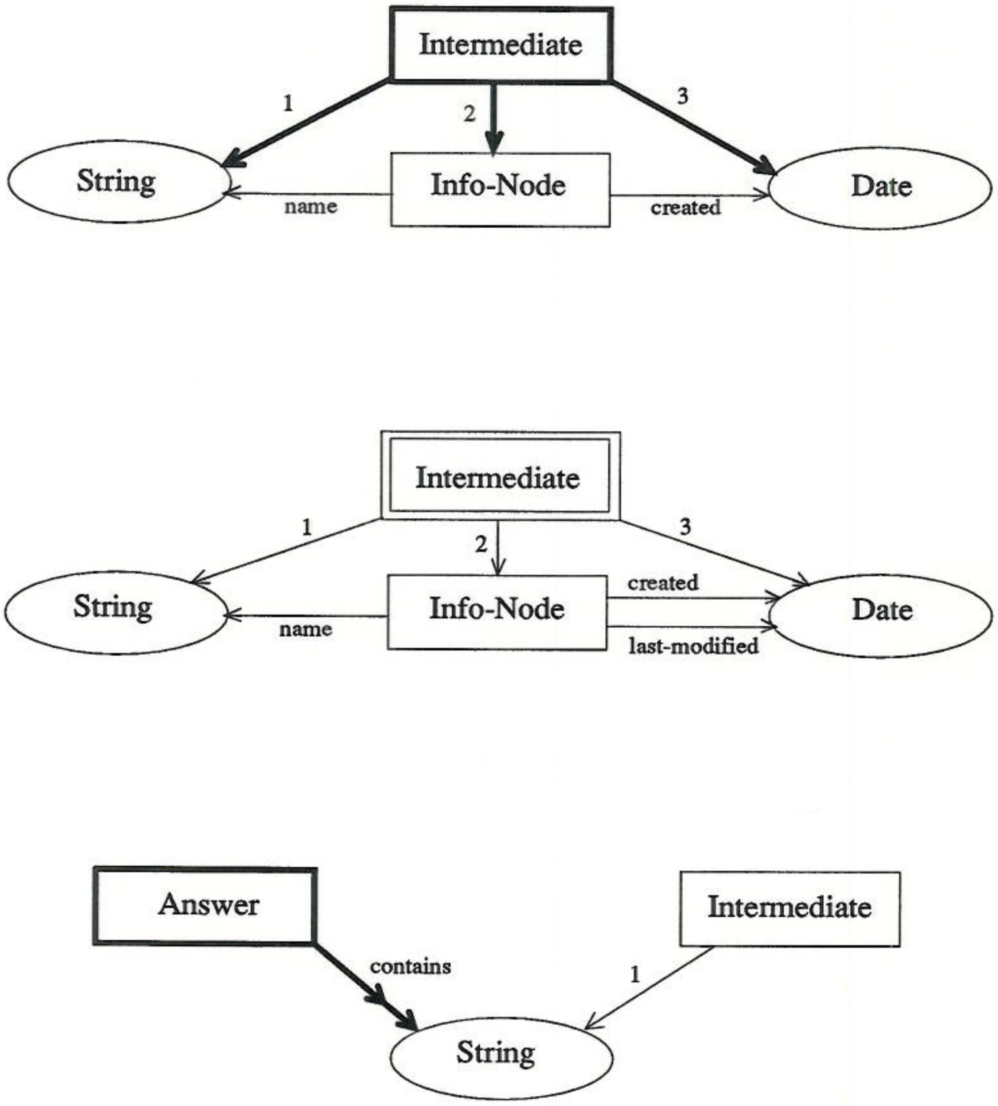


Figure 33: Simulation of negation in *GOOD*.

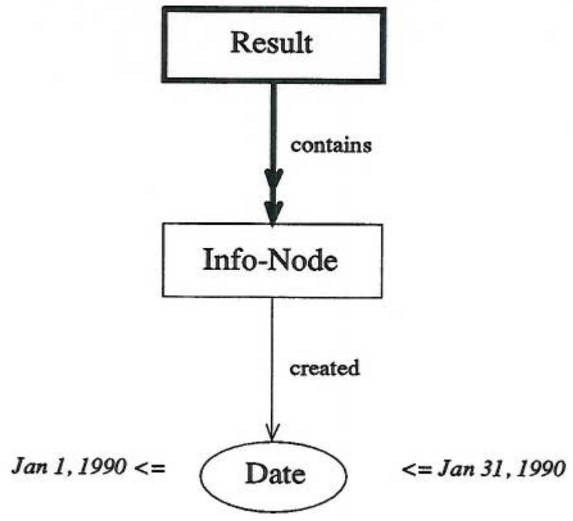


Figure 34: A range query

are deleted that are associated to an embedding that can be enlarged to the complete pattern. The intermediate nodes that are left represent the desired embeddings.

4.3 Ranges in patterns

Frequently, queries involve 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 usage of (external) functions or predicates specified over printable objects would allow one to handle this query as shown in Figure 34.

4.4 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 35 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 36 simulates the recursive edge addition of Figure 35. The first operation in the method body uses the given pattern 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

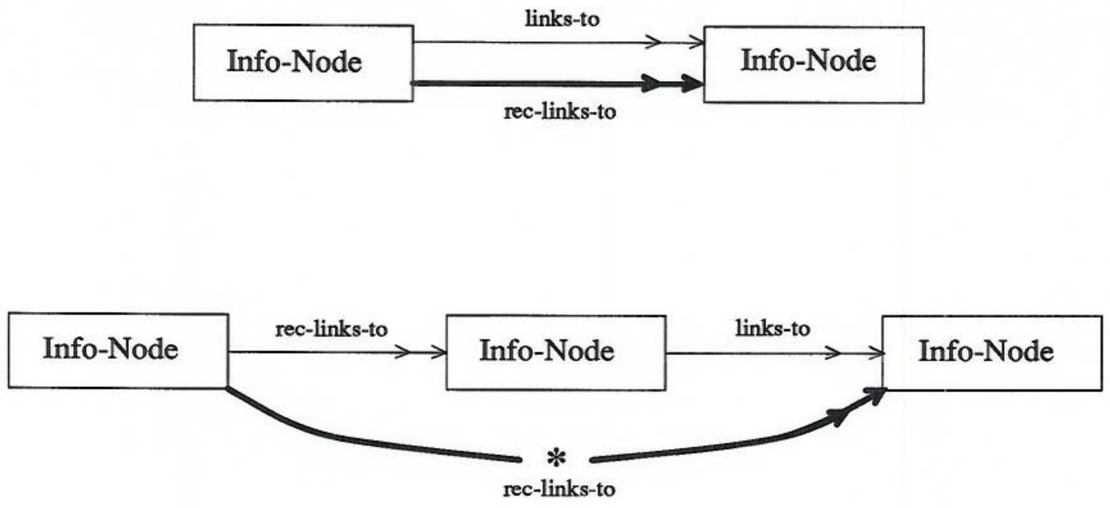


Figure 35: Computing transitive closure using recursive edge addition.

operation: this expresses the stopping condition for the recursion.

4.5 Regular expressions in patterns

The edges in a pattern can be labeled by regular expressions representing sets of strings that start and end with an edge label and that alternate edge and node labels. An edge label a can also be inverted to a^{-1}. For example, Figure 37 shows the query : "Associate to each info-node I the name of the info-nodes to which I is recursively linked, or to which I recursively refers". There is a general technique to simulate regular expressions in *GOOD*, which employs recursive edge addition.

4.6 Filters

A *filter* only consists of a pattern, depicted in a rectangle labeled "filter". Its effect is the removal of all nodes and edges that are not in the image of an embedding of the pattern. The general technique to simulate a filter in *GOOD*, is to create an intermediate node for every embedding of the pattern in the same way as for the generalized operations. The absence of these intermediate nodes is then used to delete the necessary nodes and edges.

4.7 Updates

Updates are frequently employed operations in database management. It is therefore very desirable to have a macro for updates. A straightforward suggestion for such a macro is given in Figure 38, for the update of the last-modification date of the info-node with name *Music History* to Jan 16, 1991. As already shown in Section 3.6, updates can be expressed using methods.

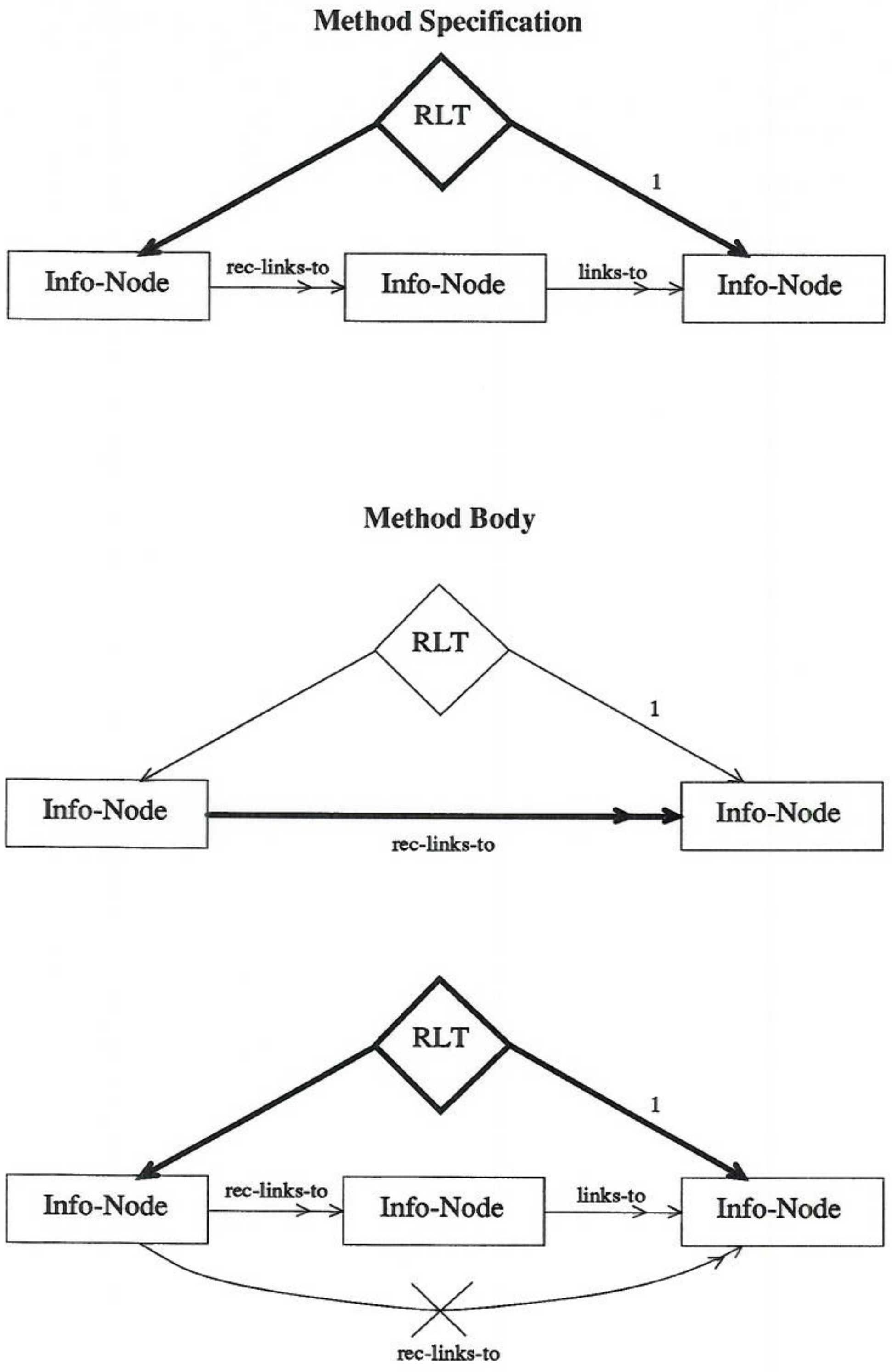
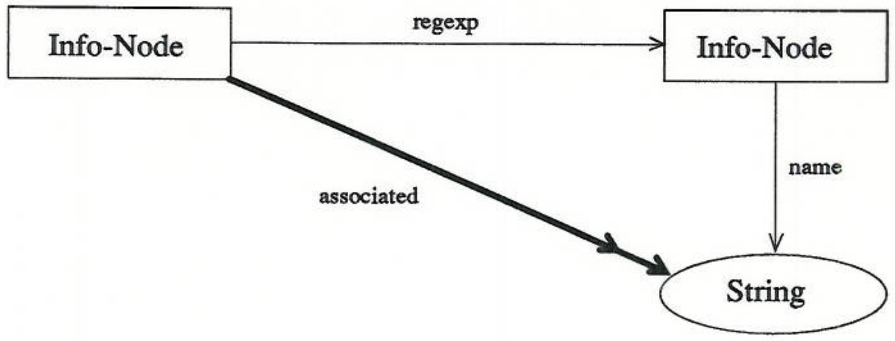


Figure 36: Simulation of recursion in *GOOD*.



$$\text{regexp} \equiv \text{links-to.(Info-Node.links-to)}^{*} \cup \text{in}^{-1}\text{.Reference.isa. (Info-Node.in}^{-1}\text{.Reference.isa)}^{*}$$

Figure 37: Regular expressions in patterns.

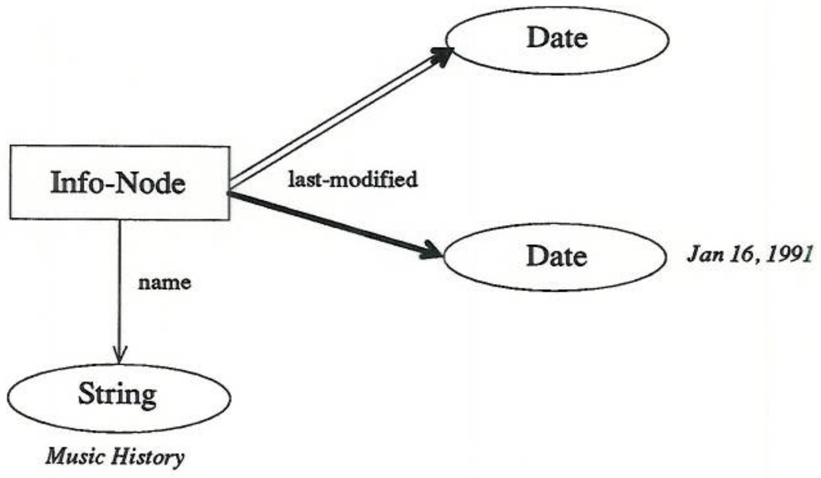


Figure 38: An update.

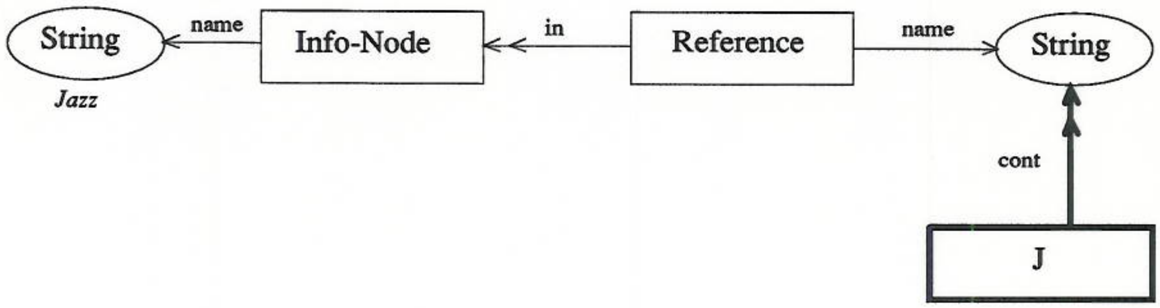


Figure 39: A *GOOD* query utilizing inheritance.

5 *GOOD* as an object-oriented data model

It turns out that *GOOD* is well-suited as a basis for an object-oriented database model. To illustrate this, we show how *GOOD* can account for some object-oriented features, as described in [6].

5.1 Complex objects and object-identity

Complex objects are typically built from primitive objects [1, 6] according to certain object constructors, such as tuples, sets and lists. Clearly, *GOOD* supports such complex objects.

The notion of *object-identity* refers to the existence of objects in the database independent of their associated properties. As stressed from the outset, object-identity is a basic feature of *GOOD*.

5.2 Inheritance

All object-oriented databases support some form of *inheritance*, i.e., it is customary to define new classes as subclasses of existing ones (e.g., [23, 36]), therefore organizing the classes in a *class hierarchy*.

In the *GOOD* model, classes can be associated with object labels in schemes. Functional edge labels can then support the notion of subclass. However, it is clear that not *all* functional edge labels in an object base scheme can be interpreted as a subclass-relationship. Therefore, we will mark the functional edges in the scheme graph we wish to interpret as subclass edges (we will implicitly assume that the subclass edges do not form a "cycle" in the object base scheme).

For example, we can consider the *isa* edges in the hyper-media object base of Figures 1–3 between *References* and *Info-Nodes* as subclass edges. The effect to the user is the same as if all properties of *Info-Node*-objects were also attached to the corresponding *Reference*-objects. The user can now apply *Info-Node* operations directly to *References*. For example, in order to obtain all references to Jazz, the user may specify the query of Figure 39 (observe that this is a generalized addition, as introduced in Section 4). Since

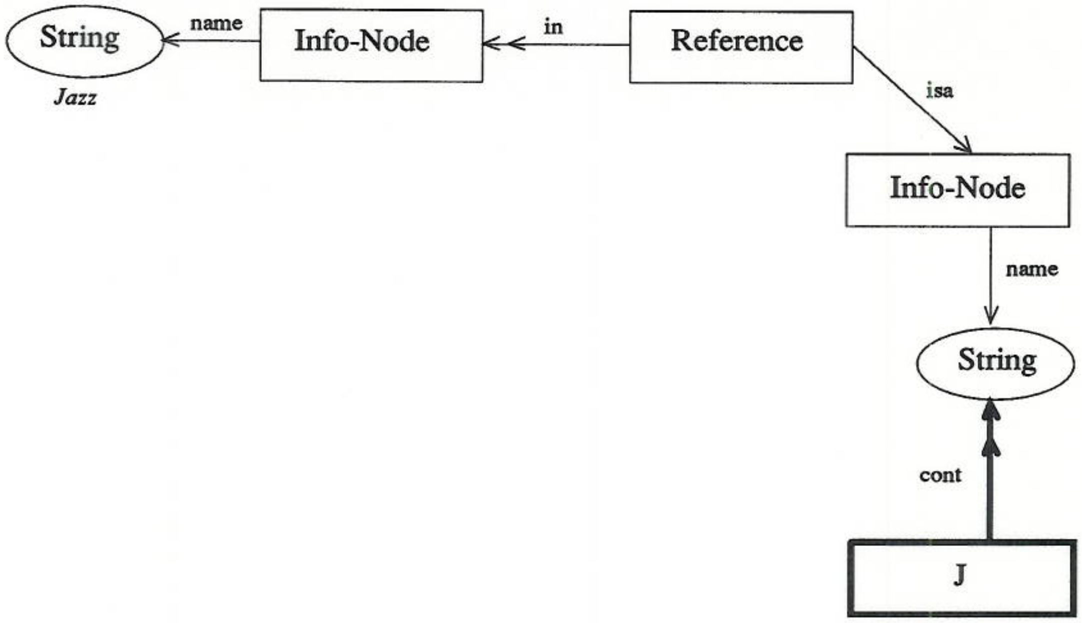


Figure 40: Simulation of inheritance in *GOOD*.

name is not a property of *Reference*, *GOOD* will translate this query internally into the query of Figure 40. 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 expressed by a method the body of which consists of a number of consecutive edge additions. Hence 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.

5.3 Method interfaces and encapsulation

From a user's perspective, methods as defined in Section 3.6 have two major disadvantages:

1. In order to understand the effect of a method, the user has to examine the method body. Needless to say, this can be a very painful job, especially if the method body contains intermediate operations manipulating information that is not explicitly known to the user.

2. Methods can have *side effects*. In order to compute the desired effect of the method, intermediate operations in the method body will generally have to introduce some "temporary" nodes and edges, which are irrelevant to the final result.

In the examples exhibited thus far, these superfluous nodes and edges were eventually deleted in the final part of the method body. As a general practice however, this solution is unacceptable. One can easily forget to delete a side effect, which can have far-reaching consequences, especially since method bodies can contain (other) method calls.

Both problems can be dealt with by modifying the definition of a method. Apart from the method specification and the method body, we add a third component: the *method interface*. The method interface is a (usually small) scheme which, intuitively describes the desired effects of the methods. More formally, Let $\mathcal{S}$ be a scheme, $\mathcal{I}$ an instance over $\mathcal{S}$ and $\mathcal{M}$ a method with interface $\mathcal{R}$. Let $\mathcal{S}'$ and $\mathcal{I}'$ be the scheme and instance resulting from applying the method $\mathcal{M}$ to $\mathcal{S}$ and $\mathcal{I}$ in the sense of Section 3.6, i.e., without taking into account the interface. In our modified version of methods, the resulting scheme $\mathcal{S}''$ is now the *union* of $\mathcal{S}$ and $\mathcal{R}$ (i.e., the smallest graph of which both $\mathcal{S}$ and $\mathcal{R}$ are subgraphs) and the resulting instance $\mathcal{I}''$ is the *restriction* of $\mathcal{I}'$ to $\mathcal{S}''$ (i.e., the largest subinstance of $\mathcal{I}'$ which is an instance over $\mathcal{S}''$).

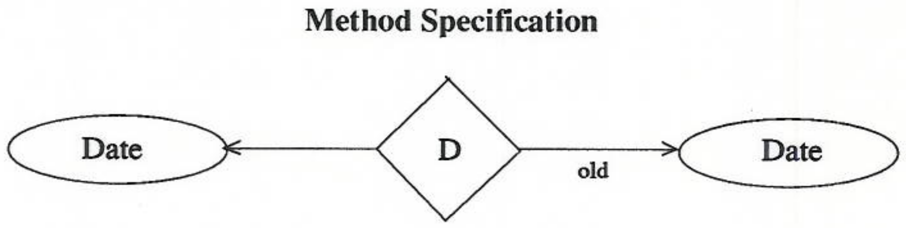
Observe that the object base scheme resulting from a method call can now be computed without any knowledge of the method body. For instance, consider a method to compute the number of days between two given dates, the specification and interface of which are shown in Figure 41. Given the user knows the meaning of the labels used in Figure 41, 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 since its last modification (Figures 42 and 43). Observe that the *Elapsed*-nodes, the introduction of which can be seen as a side 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.

An important property of object-oriented methods is that they provide *encapsulation*. *GOOD* methods with interfaces provide encapsulation, in the sense that the scheme of the result only depends on the original scheme and the method interface. Hence knowledge of the method body is not needed and unwanted side effects can be avoided.

5.4 Computational completeness

Programs in *GOOD* are built from the five basic operators, node (edge) addition, node (edge) deletion and abstraction, and the compound construct of methods. When we restrict the language to use only node (edge) addition and node (edge) deletion, we obtain a language capable of expressing the relational algebra [28]. By adding abstraction, we are as expressive as the nested relational algebra [28], and stay within PTIME complexity. Finally, the full language with methods is sufficiently strong to simulate



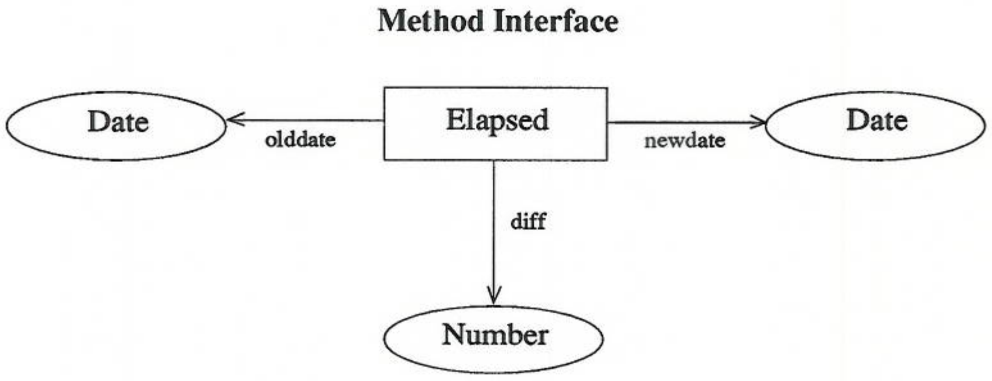
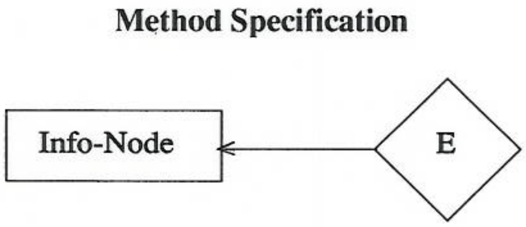


Figure 41: Specification and interface of a method to compute the number of days elapsed between two dates.



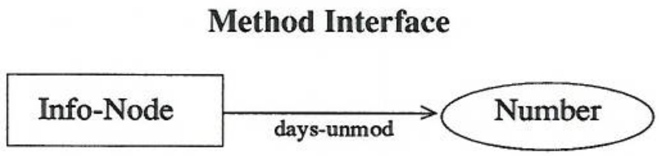


Figure 42: Method specification and interface of a method to compute the number of days elapsed since the last modification of an info-node.

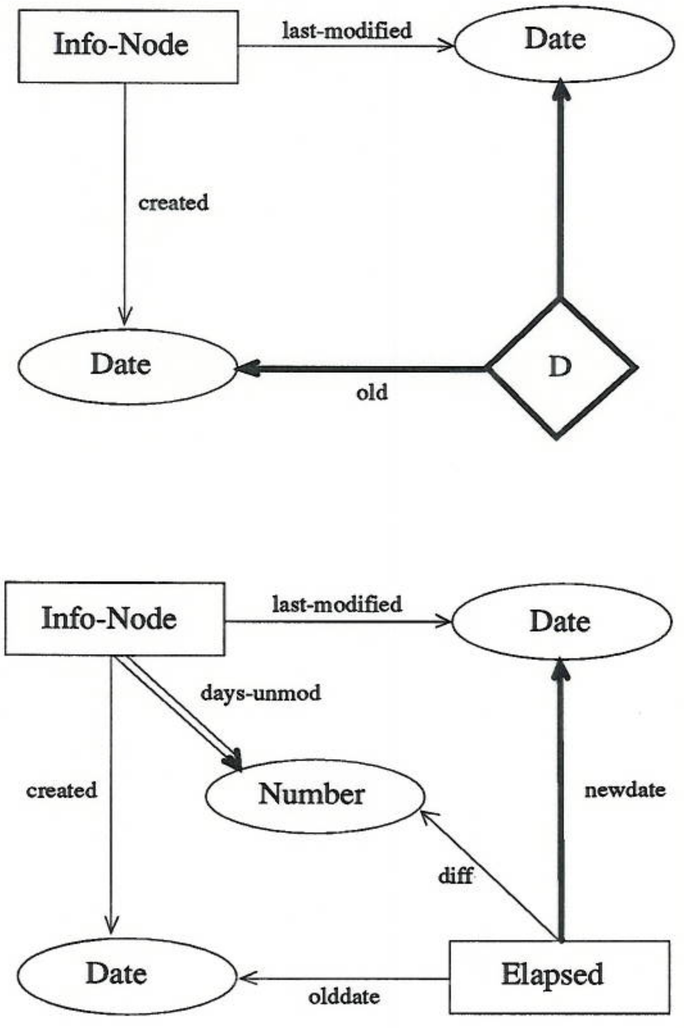


Figure 43: Method body of a method to compute the number of days elapsed since the last modification of an info-node.

arbitrary Turing Machines, as can be shown using results from [18, 34]. In fact, it can be shown that the *GOOD* transformation language is computationally complete "up to copy elimination" [3, 22], a notion raised in the recent literature which seems to be inherent to languages that employ generic creation of objects.

6 Discussion

Although *GOOD* programs are written in a procedural way, the basic operations node (edge) addition (deletion) have a declarative nature. Indeed, the pattern of such an operation can be seen as the 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 provides a basis for the development of graph-based, declarative, object-oriented database languages.

Another direction for future work is toward a better understanding of the expressiveness of the *GOOD* transformation language. As pointed out in Section 5.4, the language is computationally complete "up to copy elimination". However, in certain cases, it appears that copy elimination is not really needed. This occurs for example when the input database is ordered, or when the output is disjoint from the input [3]. It would be interesting to characterize the transformations that can be expressed in *GOOD* exactly, without the need for elimination of copies.

Finally, at the University of Antwerp, efforts are being made to implement the *GOOD* system. A detailed account on the design of a concrete user interface based on *GOOD* is given in [4]. We will couple this interface with a relational database system, allowing for a translation of the *GOOD* transformation language into a powerful query language such as Prolog or embedded SQL. Details on this translation will be given in a forthcoming paper.

References

[1] S. Abiteboul and Beeri. C. On the power of languages for the manipulation of complex objects. Technical report, INRIA, 1988.

[2] S. Abiteboul and R. Hull. Data functions, datalog and negation. In *Proceedings ACM SIGMOD Conference*, pages 143–153. ACM Press, 1988.

[3] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM-SIGMOD 1989 Annual Conference, Portland, Oregon*, pages 159–173, 1989.

[4] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. A graph-oriented user interface for object databases. Technical report, University of Antwerp (UIA), 1991.

[5] F. Arefi, C. Hughes, and D. Workman. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3):349–360, 1990.

[6] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. To be published.

[7] F. Bancilhon. Object-oriented database systems. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems, Austin*, pages 152–162, 1987.

[8] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *Proc. 2nd Int'l Workshop on Database Program. Languages*, pages 93–111, 1989.

[9] D. Bryce and R. Hull. SNAP: A graphics-based schema manager. In *Proc. of the Int'l Conf. on Data Engineering*, pages 151–164, 1986.

[10] M.J. Carey, (Ed). Special issue on extensible database systems. *Database Eng.*, June 1987.

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

[12] M. Consens and A. Mendelzon. GraphLog: A visual formalism for real life recursion. In *Proceedings Ninth ACM Symposium on Principles of Database Systems*, pages 404–416, 1990.

[13] K.F. Cruz, A.O. Mendelzon, and P.T. Wood. A graphical query language supporting recursion. In *Proc. SIGMOD Annual Conf., San Francisco*, pages 323–330, 1987.

[14] J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. Technical Report 90-23, University of Antwerp (UIA), 1990.

[15] D. Fogg. Living in a database. In *Proceedings of ACM-SIGMOD 1983 International Conference on Management of Data, Boston*, pages 100–106, 1984.

[16] A. Goldberg and D. Robson. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1985.

[17] K.J. Goldman, S.A. Goldman, P.C. Kanellakis, and S.B. Zdonik. ISIS: Interface for a semantic information system. In *Proc. of SIGMOD Conf., Austin*, pages 328–342, 1985.

[18] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In *SIGACT-SIGMOD-SIGART Symp. on Princ. of Database Systems*, 1990.

[19] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model for database end user interfaces. In *Proceedings of ACM-SIGMOD 1990 Annual Conference, Atlantic City, New Jersey*, 1990.

[20] S. Heiler and A. Rosenthal. G-WHIZ, a visual interface for the functional model with recursion. In *Proc. 11th Int'l Conference on VLDB, Stockholm*, pages 209–218, 1985.

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

[22] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Databases.* Morgan Kaufmann, 1990.

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

[24] R. King. Sembase: A semantic DBMS. In *Proc. of the First Intl. Workshop on Expert Database Systems*, pages 151–171, 1984.

[25] R. King and S. Melville. The semantics-knowledgeable interface. In *Proc. 11th Int'l Conference on VLDB, Singapore*, pages 30–37, 1984.

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

[27] A. Motro, A. D'Atri, and L. Tarentino. The design of KIVIEW an object-oriented browser. In *Proc. 2nd Int'l Conf. on Expert DB Systems*, pages 73–106, 1988.

[28] Jan Paredaens, Paul De Bra, Marc Gyssens, and Dirk Van Gucht. *The Structure of the Relational Database Model.* Number 17 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1989.

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

[30] D. Reiner, M. Brodie, G. Brown, M. Chilenskas, D. Kramlich, J. Lehman, and A. Rosenthal. The database design and evaluation workbench (DDEW). *IEEE Data. Eng.*, 7(4), 1984.

[31] D. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[32] M. Stonebraker, editor. *Readings in Database Systems.* Morgan Kaufmann Publisher, Inc., 1988.

[33] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 1,*. Comp. Science Press, 1989.

[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 *Proc. 11th Int'l Conference on VLDB, Mexico City*, pages 22–32, 1982.

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

[37] M. Zloof. Query-By-Example. *IBM Syst. Journal*, 16:324–343, 1983.