An Overview of GOOD

Jan Paredaens Jan Van den Bussche[a]
Marc Andries Marc Gemis Marc Gyssens[b] Inge Thyssens
University of Antwerp (UIA), Dept. Math. & Comp. Science, B-2610 Antwerp, Belgium
Dirk Van Gucht Vijay Sarathy Lawrence Saxton[c]
Indiana University, Computer Science Department, Bloomington, IN 47405, USA

[a]Aspirant NFWO.
[b]University of Limburg, Dept. WNIF, B-3590 Diepenbeek, Belgium.
[c]University of Regina, Dept. Computer Science, Regina, Saskatchewan S4S0A2, Canada.

Abstract

GOOD is an acronym, standing for *Graph-Oriented Object Database.* GOOD is being developed as a joint research effort of Indiana University and the University of Antwerp. The main thrust behind the project is to indicate general concepts that are fundamental to any graph-oriented database user-interface. GOOD does not restrict its attention to well-considered topics such as ad-hoc query facilities, but wants to cover the full spectrum of database manipulations. The idea of graph-pattern matching as a uniform object manipulation primitive offers a uniform framework in which this can be accomplished.

1 The GOOD model

In this section, we informally introduce the basic model of GOOD. For a complete treatment, see [1, 2, 3].

Basic to any graphical database user-interface is the visualization of the database *scheme.* In GOOD we take an approach as general and simple as possible, considering a scheme to be a directed, labeled graph whose nodes represent classes of objects and whose edges represent relationships or properties that can exist between objects of these classes. For example, Figure 1 shows the scheme for a hypermedia system, storing documents that may contain text, graphics and sound information. A rectangular node (like `Text`) represents an abstract class; an oval node (like `Long-String`) represents a basic class. The `#chars` edge indicates the number of characters in a text. Note that `Reference`, `Sound`, `Text` and `Graphics` are subclasses of `Info-Node`. They inherit all its properties.

The key idea in GOOD is that even a database *instance* can be seen as a graph (at least conceptually). In this graph, each object and each value is represented by a unique node, in accordance with the paradigm of object identity. Furthermore, each such node is labeled by a class name occurring in the database scheme; basic class nodes are additionally labeled by their value. Finally, the edges in the instance graph stand for the various relationships or properties between the objects in the instance and must be labeled conforming to the scheme, i.e., for each edge in the instance graph there must be a corresponding edge in the scheme graph having the same labels for the source node, target node, and the edge itself. Due to space limitations, we omit an instance graph example.

It is typical in graph-based database user-interfaces to express queries visually by graphs which are built from components of the scheme graph. The actual structure of this query graph specifies which portions of the database are to be retrieved by the query. In GOOD, the function of the query graph is abstracted in the concept of *pattern,* which will serve as a natural, uniform primitive for all database manipulations (not only querying).

A pattern is very similar to an instance graph. Syntactically, the only difference is that in an instance graph the basic class nodes are labeled by their values, whereas, in a pattern this is not required. Also semantically, the pattern stands for a "sample" of (a part of) the instance. Informally, applying a pattern to an instance results in a number of matchings, each of them corresponding to a subgraph of the instance that corresponds to the pattern. Formally, a *matching* of a pattern J in an instance I is a mapping from the nodes of J to the nodes of I, preserving all edges between the nodes, and all labels. For example, in Figure 2, two patterns are shown. Each matching of the leftmost pattern represents an Info-node with name 'Reggae' together with an Info-node to which it is linked, and the creation date of the latter. Each matching of the rightmost pattern stands for an Info-node with one of its references that refers to the node itself. (It may be interesting to note that, since patterns are syntactically database instances, they can be treated (e.g., stored) in the same way as "real" instances. The objects of a pattern can be thought of as "computational" or "virtual" objects.)

We now come to the operations of the GOOD model. At the conceptual level, every GOOD operation has the effect of a graph transformation. Operations consist of a pattern together with an action. When applying the operation to an instance, the action is performed on each matching of the pattern, in parallel. More concretely, there are five basic operations defined in GOOD: *node addition*, *edge addition*, *node deletion*, *edge deletion*, and *abstraction*. Furthermore, there is a construct for grouping operations in *methods*. Here, we will only explain node/edge addition/deletion. For abstraction, we refer to [4]; for methods, see [2, 3].

The node addition serves to add data, be it as derived data in a query or as an update, in the form of new nodes in the instance graph, with outgoing edges ending in objects that already exist. Syntactically, a node addition has the form shown in Figure 3 (top). The pattern specifies the places in the instance where new nodes are to be added. The pattern is augmented with an additional bold node and outgoing edges. This means that for each matching of the pattern, such a new node and corresponding edges are added. For example, the node addition of Figure 3 has the effect that for each pair of Info-nodes (the first of which having name 'Rock') an object of class *Pair* is created with an attribute-edge named *parent* for the creation date of the first object and an attribute-edge named *child* for the creation data of the second object.

The edge addition serves to add data in the form of new edges in the instance graph. Syntactically, an edge addition has the form shown in Figure 3, bottom. The pattern specifies the places in the instance where new edges are to be added. The pattern is augmented with one or more bold edges. This means that for each matching of the pattern, the corresponding edges are added. For example, the edge addition of Figure 3 has the effect that the creation date of the 'Pinkfloyd' Info-node is associated with all Text-nodes linked to it.

Both addition operations have an addition as effect. There are two corresponding, complementary operations, the node and edge deletion, whose effect is a deletion, be it as an update, or be it in a query to express negation. They are syntactically specified by a pattern, in which certain nodes or edges are indicated in double outline. This means that in each matching of the pattern, the corresponding nodes or edges have to be removed. Space limitations prevent us from giving an example.

Using the GOOD operations the user can express almost all computable transformations on the instance graph. As such he specifies a sequence of operations to be applied to the database. Such a sequence is called a GOOD-program.

2 The GOOD system

In this section, we briefly explain how the GOOD model can be used in practice and sketch the state of affairs in implementing the GOOD system at the University of Antwerp. The work is divided into two major parts: the *user-interface*, which fully supports the GOOD database language as the uniform means by which the user specifies the database tasks to be performed by the system, and the *database management*, part which supports the link between the user-interface and the actual DBMS.

2.1 The GOOD user-interface

Initially, an experimental prototype of a GOOD-based user-interface was developed on a Macintosh [5]. Based on this experience we decided to develop a full-fledged user-interface for the GOOD system. New design decisions were made, some of which are reported in [7]. The actual implementation efforts are currently on-going: the program is built under

X Window, using OSF/Motif.[1]

Naturally, the central component of the user-interface is the graphical representation of the database scheme, which contains all relevant "syntactic" information. There are two simple, yet powerful facilities [7] that allow the user to alter the specific graphical representation used to depict the scheme. By *decomposing* the scheme graph on a particular class node, the fragment of the scheme concerned with that class is disconnected from the rest of the graph. Decomposition can be undone by the corresponding *compose* facility. Using composition and decomposition, the user's perspective on the database scheme can be altered flexibly.

The scheme graph is important since it is the basic means by which the user can specify patterns. We discussed how syntactically a pattern is a graph that conforms to the structure of the scheme. Alternatively, a pattern can be seen as being "generated" by the scheme graph. This last observation is exploited in the user-interface: the user assembles a pattern simply by copying, duplicating and identifying nodes and edges from the scheme graph. An advantage of this approach is that it is *syntax-directed* in a natural, graph-oriented manner. Indeed, it is impossible to construct illegal patterns, and hence almost all syntactical and many semantical errors are avoided.[2] Moreover, taking into account the pattern-matching semantics of GOOD, we obtain a mode of user interaction compatible with the *direct manipulation* paradigm for object manipulation. Indeed, as the user builds a pattern-operation, he can actually think of this pattern as a "sample" of the database with which he is working. Important for the bulk-processing nature so typical for database applications is that in reality, *all* matchings of the pattern are considered.

Once patterns can be constructed, arbitrary GOOD programs can be built. This is implemented in the standard Motif user-interface look and feel. Actually, our user-interface understands a superset of the basic GOOD language described earlier. Indeed, several *macros*, consisting of frequently needed sequences of operations, and slight extensions of the pattern concept to deal with arithmetic and the like are supported [3, 6].

[1] OSF/Motif is a trademark of the Open Software Foundation.

[2] Here, *syntax* must be interpreted broadly, since our programs are expressed as graphs, not texts.

Finally, we mention that we have developed two rather novel mechanisms, *viewing* and *browsing* [7], which allow the user to interact with the real data. These two tools are basically meant to inspect the result of a GOOD program (see below for a discussion of what this result exactly is). However, these mechanisms can as well be used for simple ad-hoc querying (viewing) and for navigational object-centered access (browsing). Viewing and browsing are again uniformly based on the concept of patterns. For example, we define browsing as an extension of pattern matching where certain nodes of the pattern are constrained to be mapped to specific, predetermined objects in the instance.

2.2 Database management in GOOD

Up to now, the effect of a GOOD program was formally considered to be a general transformation of the input database into the output database. However, in practice, the actual effect of this transformation on the stored database is determined by the one out of five different *modes* in which the user elects to run his program [5, 7]. Each mode interprets the formal transformation in its own particular way, both on the schema and on the instance level, either as a *restructuring*, a *query*, a *constraint specification*, an *update*, or a *schema modification*.

Once the operations to be performed on the real data have been determined from the running mode, they must be executed by a concrete DBMS. For this purpose, we store a GOOD database, both whose scheme and instance are graphs only on the *conceptual* level, in a relational DBMS. Consequently, the algorithms for pattern matchings and the corresponding GOOD operations are translated into a relational DML. Of course, we must also support the other direction of this translation, since the result of the produced relational transaction must be interpreted back to the graph-oriented viewing and browsing tools described earlier.

Since we do not want to be dependent upon one particular DML, we have formally specified an relational database abstract machine, listing the capabilities that GOOD requires of any underlying DBMS. Then all database commands issued by the GOOD system call only upon this abstract machine.

3 Future research directions

3.1 Implementation of GOOD in the Tarski Database Model

As discussed in Section 2.2 a natural way to implement GOOD is on a relational database abstract machine. This strategy however deviates from the main philosophy of the GOOD model, which is to specify all database objects and processes in terms of graphs. To remain close to this philosophy, we developed the Tarski database model (TDM) [8]. In the TDM, all data is represented in (untyped) binary relations (i.e., graphs) and all processes are formulated as expressions in the (extended) Tarski algebra. The core of this algebra is an adaptation of an algebra of Tarski and Givant[3] to the domain of *finite* binary relations. The core has six operators. Of those, the union ($r \cup s$), the relational composition ($r \cdot s$), the inverse (r^{-1}), and the (finite) complement ($\bar{r}$) are well-known. In addition, there are two *tagging* operators, the left-tagging operator ($r^{\triangleleft}$) and the right-tagging operator ($r^{\triangleright}$). These operators associate unique tags to the ordered pairs in a relation (see Figure 4) [4]. As shown in [8], this core of the (extended) Tarski algebra is as expressive as the Codd-relational algebra. The adjective "extended" indicates that the core algebra is extended with standard programming language constructs such as variables and assignment, compound, if-then-else, and iterative statements.

The Tarski database model allows for graph-oriented support of GOOD instances at the *physical* level. Let us illustrate this with an example. Consider the persons database shown as a GOOD instance in Figure 5 (top). In Figure 5 (middle) we show a corresponding physical representation as a set of binary relations. In this representation, there is a separate relation for each node type (except for the atomic node types) and there is a separate relation for each edge type[5]. Now consider the edge addition and node addition operations shown in Figure 5 (bottom). The edge addition can be performed in the Tarski algebra with the assignment statement $grandparent := parent \cdot parent$ and the node addition can be performed by the statements [6]:

$fatherPerson := child^{l\pi} \cdot (sex \cdot \{(\mathtt{male},\mathtt{male})\})^{l\pi};$
$isPerson := fatherPerson^{\triangleleft};$
$Father := isPerson^{l\pi}$

It might be objected that Tarski algebra expressions are complex and difficult to interpret. This is indeed the case. However, this doesn't render the algebra an uninteresting target language. In fact, work on the *decomposed storage model*[7] points to advantages of the Tarski relational approach relative to the Codd relational approach, especially in the context of parallel computation. Therefore, we are currently coupling the GOOD model, via the Tarski data model, to the decomposed storage model. Within this scope, an interesting research avenue will be the study of query optimization of GOOD data manipulation expressed as equivalent (extended) Tarski algebra expressions.

3.2 GOOD and graphical interfaces

Section 2.1 describes an implementation effort of the GOOD user-interface that corresponds closely to the basic design of the GOOD model [1, 2, 3]. Although this interface exhibits high-level graphical features, there exists database applications that are not "naturally" modeled within it.

To stay within the realm of database applications involving graphs, consider a *recursive roadmap*. At the top level such a roadmap might be a graph with the major cities of the United States as nodes and with the main direct routes between these cities as edges. In addition, the recursive roadmap has two special features: 1) most of the interesting data (such as type of route, distance in miles, distance in kilometers) is associated directly to the *edges* (routes) of the roadmap, and 2) the roadmap is

[3]A. Tarski and S. Givant, *A formalization of set theory without variables*, American Mathematical Society, 1986.

[4] As pointed out by Ore (*Theory of Graphs*, American Mathematical Society, 1962), there is a natural interpretation of the first four operators in terms of graphs manipulations (for example, $\bar{r}$ corresponds to computing the complement-graph of r viewed as a graph). (These tags serve the role of object identifies for the ordered pairs.) The left and right tagging operation on a relation r can best be understood in terms of a GOOD node addition.

[5]Clearly, this is not the only reasonable representation of this GOOD database as a set of binary relations

[6] In these statements the operation $r^{l\pi}$ denotes the expression $(r^{\triangleleft})^{-1} \cdot r^{\triangleleft}$ and $\{(\mathtt{male},\mathtt{male})\}$ denotes the singleton binary relation containing the pair (male,male). It should be clear that there are other programs to simulate the above GOOD operations.

[7]S. Khoshafian, G. Copeland, T. Jagodits, H. Boral, and P. Valduriez, A query processing strategy for the decomposed storage model, in *Proc. of Data Engineering Conference*, Los Angeles, CA, 1987, pp. 636–643.

recursive in the sense that a city on the roadmap might have its own associated (recursive) roadmap, being a graph of landmarks within that city and their connecting routes. Although GOOD can conceptually model this roadmap, this conceptualization is not in itself a natural graphical rendering of the (typical) roadmap. This is because 1) edges in GOOD carry no information but their labels (thus edges with "semantics" need to be modeled as nodes in GOOD and therefore loose their natural edge status), and 2) at the conceptual level, GOOD is essentially a "flat" model, i.e., in the recursive roadmap, it would model cities and landmarks at the same level of abstraction.

Our view is that, in its pure formulation, the GOOD model is a conceptual model that *facilitates* reasoning about graphical database user-interfaces. More specifically, the two-dimensional, graph-oriented way of specifying database manipulations of GOOD offers advantages over the more common one-dimensional sentence-oriented way. However, as we pointed out in the recursive roadmap example, proper conceptualization is different from natural graphical rendering. To address this issue, but keep the advantages of the GOOD approach, we are designing a database model which remains close to the basic graph-oriented philosophy of the GOOD model but which differs from it markedly in two respects. First, in this database model, there is a symmetric treatment of nodes and edges. For example, whereas in the GOOD model only nodes are in a natural way associated with properties and values, in our new database model, this becomes valid for edges as well. Second, nodes and edges have an internal organization and state that is (by default) hidden or encapsulated, but that can be revealed if needed or desired. This second feature offers the opportunity to present information at different levels of abstraction, a technique so common in current graphical user-interfaces, but under-developed or under-exploited in database user-interfaces. We plan to implement a graphical user-interface supporting this database model in the X Window OSF/Motif environment, just as is now done for the GOOD interface.

References

[1] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In *Proc. 9th PODS*, pp. 417–424. 1990.

[2] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model for database end-user interfaces. In *Proc. 1990 SIGMOD*, pp. 24–33.

[3] M. Gyssens, J. Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. Technical Report 327, Computer science department, Indiana University, 1991. Submitted to *IEEE Trans. Knowl. Data Eng.*

[4] J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In *Proc. 10th PODS*, pp. 291–299. 1991.

[5] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. A graph-oriented user interface for object databases. Technical Report 91-04, University of Antwerp (UIA), 1991.

[6] M. Andries and J. Paredaens. Macros for the GOOD Transformation language. Technical Report 91-20, University of Antwerp (UIA), 1991.

[7] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. Concepts for graph-oriented object manipulation. Technical Report 91-36, University of Antwerp (UIA), 1991. To appear in *EDBT 1992.*

[8] M. Gyssens, L.V. Saxton, and D. Van Gucht. Tagging as an alternative to object creation. Submitted.

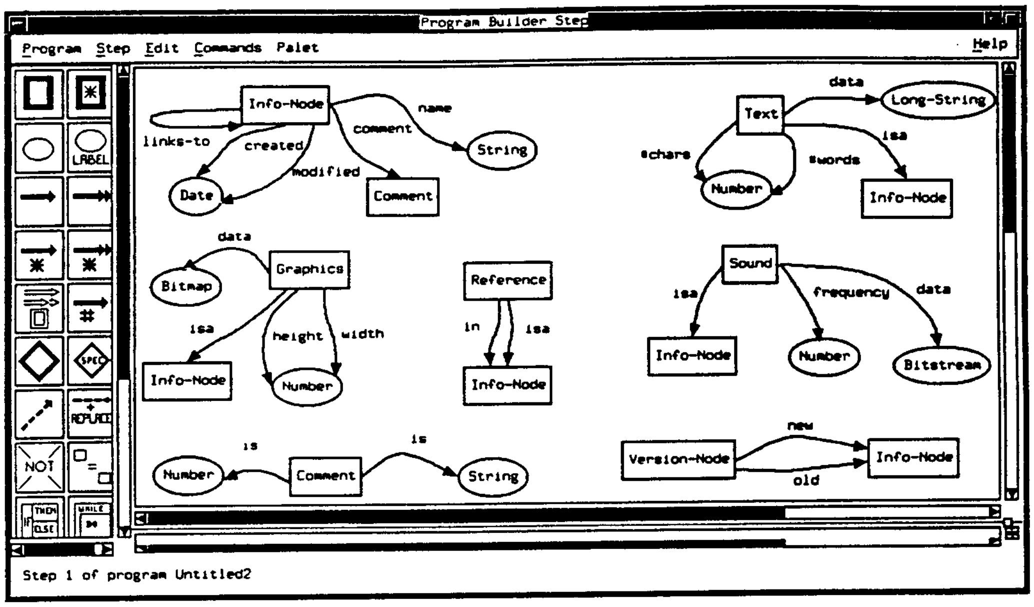


Figure 1: A GOOD database scheme.

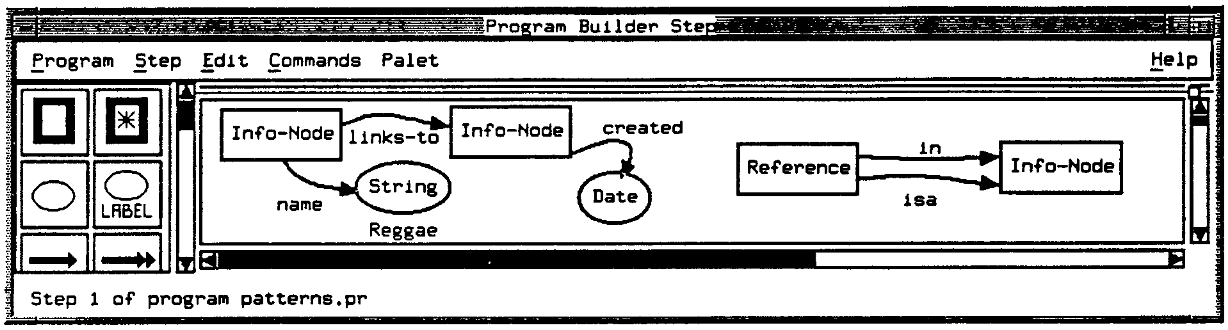
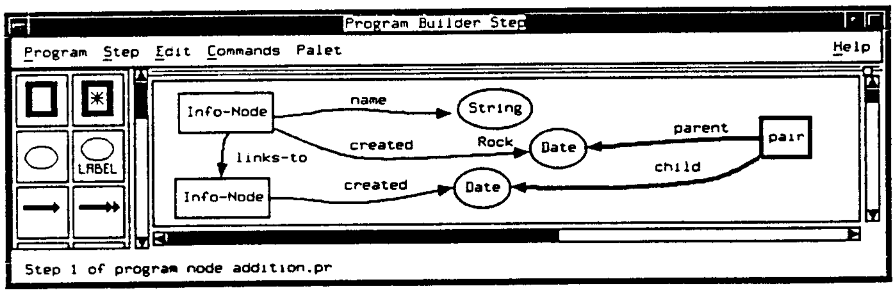


Figure 2: GOOD patterns.



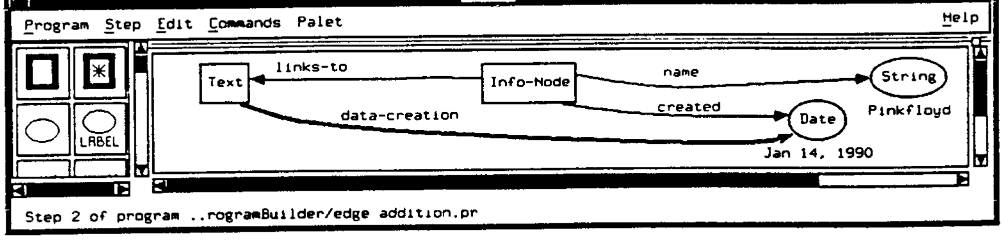

xmMain
Program Step Edit Commands Palet Help
Text
links-to
Info-Node
name
String
Pinkfloyd
data-creation
created
Date
Jan 14, 1990
Step 2 of program ..rogramBuilder/edge addition.pr

Figure 3: GOOD operations.

r

a	b
a	c
b	c

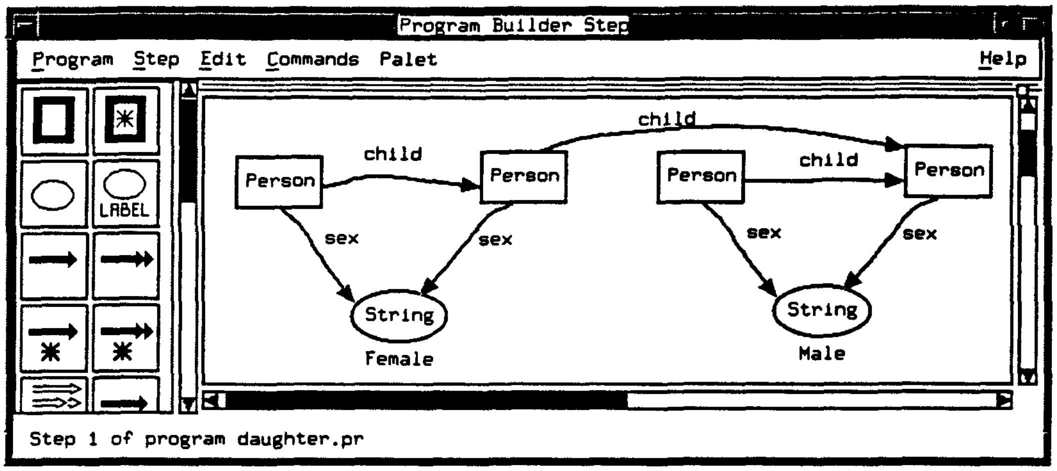
$r^{\triangleleft}$

1	a
2	a
3	b

$r^{\triangleright}$

1	b
2	c
3	c

Figure 4: Example of left and right tagging a relation. The tag 1 is a succinct representation of the ordered pair (a, b).



Step 1 of program daughter.pr

Person

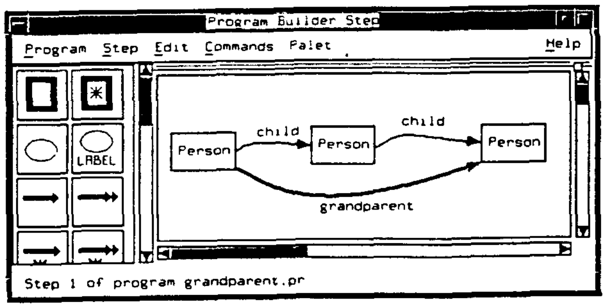
p_1	p_1
p_2	p_2
p_3	p_3
p_4	p_4

child

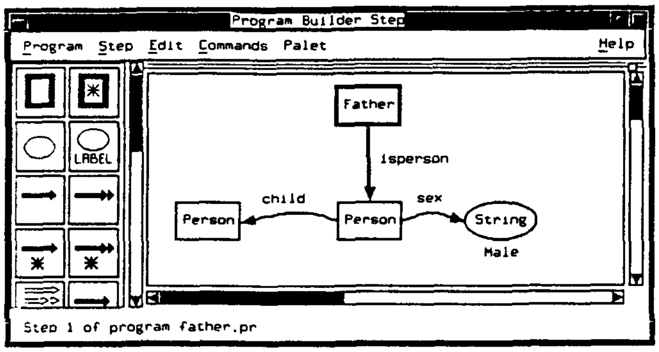
p_1	p_2
p_2	p_4
p_3	p_4

sex

p_1	*Female*
p_2	*Female*
p_3	*Male*
p_4	*Male*



Step 1 of program grandparent.pr



Step 1 of program father.pr

Figure 5: A persons database as a GOOD instance (top). A representation of the persons database as a set of binary relations (middle); the values p_1, p_2 etc. denote unique person identifiers. A GOOD edge addition and a GOOD node addition (bottom)