A Visual Interface for a Database with Version Management

JAY W. DAVISON
AT&T Bell Laboratories
and
STANLEY B. ZDONIK
Brown University

This paper describes a graphical interface to an experimental database system which incorporates a built-in version control mechanism that maintains a history of the database development and changes. The system is an extension of ISIS [6], Interface for a Semantic Information System, a workstation-based, graphical database programming tool developed at Brown University. ISIS supports a graphical interface to a modified subset of the Semantic Data Model (SDM) [7]. The ISIS extension introduces a transaction mechanism that interacts with the version control facilities.

A series of version control support tools have been added to ISIS to provide a notion of history to user-created databases. The user can form new versions of three types of ISIS objects: a class definition object (a type), the set of instances of a class (the content), and an entity. A version-viewing mechanism is provided to allow for the comparison of various object versions. Database operations are grouped together in atomic units to form transactions, which are stored as entities in the database. A sample session demonstrates the capabilities of version and transaction control during the creation and manipulation of database objects.

Categories and Subject Descriptors: H.1.2 [**Models and Principles**]: user/machine systems—*human factors*; H.2.1 [**Database Management**]: Logical Design—*data models*; H.4.1 [**Information Systems Applications**]: Office Automation

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Historical database, semantic data model, transaction processing, version control, visual interfaces

1. INTRODUCTION

An office environment is characterized by the production of artifacts that are used in the execution of office procedures. For example, an office worker will routinely generate reports that describe the progress of some project, graphical images that are used in presentations to help bolster a new idea, or a spread-

This work was supported in part by the Office of Naval Research under contract N0014-83-K-0146 and DARPA under order 4786.

Authors' addresses: Jay W. Davison, Department D-111, MITRE Corporation, Burlington Road, Bedford, MA 01730; Stanley B. Zdonik, Department of Computer Science, Brown University, Providence RI 02912; sbz@brown.csnet.

sheet that will be used in making a financial decision. We envision an office information environment in which all of these objects are stored in a powerful database facility. Artifacts such as these typically undergo continual change as the office worker refines his or her notion of what is needed.

We might say then that the office worker is involved in a kind of design activity. We believe that any such activity needs support tools for managing this change over time. Workers will need to compare past versions of an object with its most current version. Adding a time dimension potentially adds to the complexity of the system. It is important to develop new ways to provide access to this additional information such that it enhances productivity instead of obscuring the real focus of a task. The development of intelligent interfaces that are able to deal with the time dimension is an important step in this direction.

ISIS-V is a database system interface that allows users to access graphically a database that supports a high-level, semantic data model similar to the Semantic Data Model (SDM) [7]. This paper describes the design of an experimental vehicle that has been developed at Brown to test the feasibility of adding access to the time dimension in a visual database environment.

ISIS-V extends the ideas in [6] to include simple notions of versions and transactions. There has been much recent interest [5, 9, 10, 13, 17, 21] in the notion of databases that can record the intermediate states of an object as it evolves through time. It is possible to apply general version-handling operations to three types of ISIS-V objects: the class definition, the class content, and the data entity. By performing database operations that alter the definition of any of these database entities, the user can create successive versions of the object in a linear version set. These versions are stored in the database as changes to the original definition, and can be compared using ISIS-V version-viewing mechanisms.

An ISIS-V transaction is a grouping of operations that is viewed as an atomic action on the database state. ISIS-V is always operating in the context of some transaction. A user can explicitly start a new transaction and thereby end the previous one. When a transaction ends, any changes that are made by that transaction are committed to the database.

We treat transactions as objects and store them in a reserved TRANSACTION class. It is possible to access the write-set of each of these transactions. This makes is possible to access prior states of the database for viewing and comparison purposes. In this sense, ISIS-V transactions provide a means for tracking the history of the database. Completing a transaction is analogous to creating a new version of the database.

For the remainder of this document we shall use the name ISIS to refer to the ISIS-V system. If we want to refer to the predecessor of ISIS-V, we shall explicitly say "the original ISIS."

2. SYSTEM DESCRIPTION

The original ISIS [6] is an experimental, workstation-based system, exploiting a two-dimensional visual display to allow for graphical manipulation in database programming. The original ISIS provides a rich set of graphical operations to

—construct a new database, or modify an existing one,

—browse through both the schema and data levels,
—build complex queries, the results of which can be saved for future use.

2.1 Semantic Database Model: The ISIS Subset

ISIS is based on the Semantic Data Model [7]. To make the construction of ISIS more tractable, a relationally complete subset of SDM has been chosen for implementation. This modified subset expresses the essential features and reflects the basic design principles of SDM.

The three primary building blocks in the ISIS system are the *entity*, *class*, and *attribute*. An ISIS database consists of a collection of entities that are organized into classes. Entities are defined to have named attributes whose values are other entities. An ISIS *schema* consists of a set of class definitions. A class is a named collection of entities that all have the same attributes.

A class can be designated as either *base* or *nonbase*. A base class is defined independently of all other database classes. Nonbase classes are dependent on base classes and are logically related to them through *interclass connections*. ISIS contains two types of interclass connections—*subclass connections* and *grouping connections*. A subclass S of a class T is constrained to contain some subset of the membership set for T. We say that T is a *superclass* of S. A grouping class G of a class T contains sets whose members are drawn from T. Each member of a particular set in G shares a common value for one designated *grouping attribute*.

ISIS has five *predefined* base classes, of which the following four are available to the ISIS user as value classes: INTEGERS, REALS, YES/NO (Booleans), and STRINGS. The fifth predefined class, the TRANSACTION class, described in a later section, is not available for general access since it is used to store version control information.

Entities and classes in ISIS have an associated collection of attributes that are used to describe their characteristics and relate them to other classes and entities. Each attribute has a *value class* from which the values of the attribute are drawn. These *attribute values* are used to give descriptive information about ISIS entities. Attributes are either *single valued* or *multivalued*. A single-valued attribute has one value drawn from the attribute's value class, and a multivalued attribute has a set of values that constitutes a subclass of the attribute's value class. Members of subclasses have single-parent inheritance of attributes. That is, if S is a subclass of T, S automatically inherits all attributes defined on T. Inheritance is recursive, and S inherits all attributes that are defined on any superclass of T as well.

2.2 Interface Characteristics

At each level, an ISIS screen consists of a collection of disjoint windows (see Figure 4, p. 243). There are four possible window types. The largest of these windows is the work area for the current level. This rectangle typically contains the class(es) or entity(ies) under consideration at the moment. A hand with a pointed-finger icon is usually present at each level, and is used to denote the current database object. If the hand is not in view, then either there is no current object or the current object is not currently in view (in which case panning is necessary).

Menu keys appear at the bottom of every ISIS screen. The selection of menu keys varies from level to level in ISIS, and furthermore may vary according to the current object in the database. Command functions are standardized, however, so that menu keys in different levels that have the same name will also have the same semantics. Generally speaking, the first four commands alter the description of the current object, whereas the last four commands allow for traversal to different screens and levels.

Each screen also has a corresponding set of menu keys that provides access to version operations. These menu keys can be viewed by pressing the third mouse button. This puts the use of the version control mechanism totally under the user's control; if the user does not wish to use version control, no version control operations will appear. Any time that the version menu is toggled to the standard menu again, the database is automatically brought back to its most recent state, since modification of database objects can only be performed on the current database state.

The final type of window is a pop-up edit window, which provides the user with a set of graphical editing operations that can be performed on the current object. This window appears at the current cursor location when the user picks the second mouse button. The user is presented with a box, which is dragged over the menu operations until the desired operation is within the box. Picking any mouse botton at this point will perform the chosen operation on the current object. Each ISIS screen has its own graphical editing menu, since the type of operations that can be performed may vary according to the current object and its method of representation within the current screen.

2.3 ISIS Levels

ISIS views a database as consisting of two interdependent levels, the *schema level* and the *data level*, each of which contains a series of screens which allow navigation using maps formed by attribute value classes and attribute values (see Figure 1). In addition, version comparison is available at both levels through the use of an additional screen, the *version card* screen. The hand icon is present at each level and in each screen to indicate the current database object.

2.3.1 *Schema Level.* The schema level provides a view of the schema plane through the use of three different screens: *inheritance forest*, *semantic network*, and *predicate work sheet.*

Upon entering ISIS, the inheritance forest is the first screen that comes into view (see Figure 4, p. 243). This screen provides a view of the classes, groupings, and attributes that comprise the schema. Interclass connection lines connect groupings (which lie above the father class) and subclasses (which lie below) to their respective father classes. The hand icon points to the currently selected schema object. When the user toggles the menu keys to the version menu, those class types that have previous versions are redrawn with a three-dimensional box outline. The user can then create new versions of a class and view preceding and succeeding versions. The inheritance forest will be redrawn to reflect the changes performed to the schema objects as a result of the version traversal. The box outline remains for each class type that has a preceding version. As the user views past versions of a class type, a small arrow appears at the lower left corner

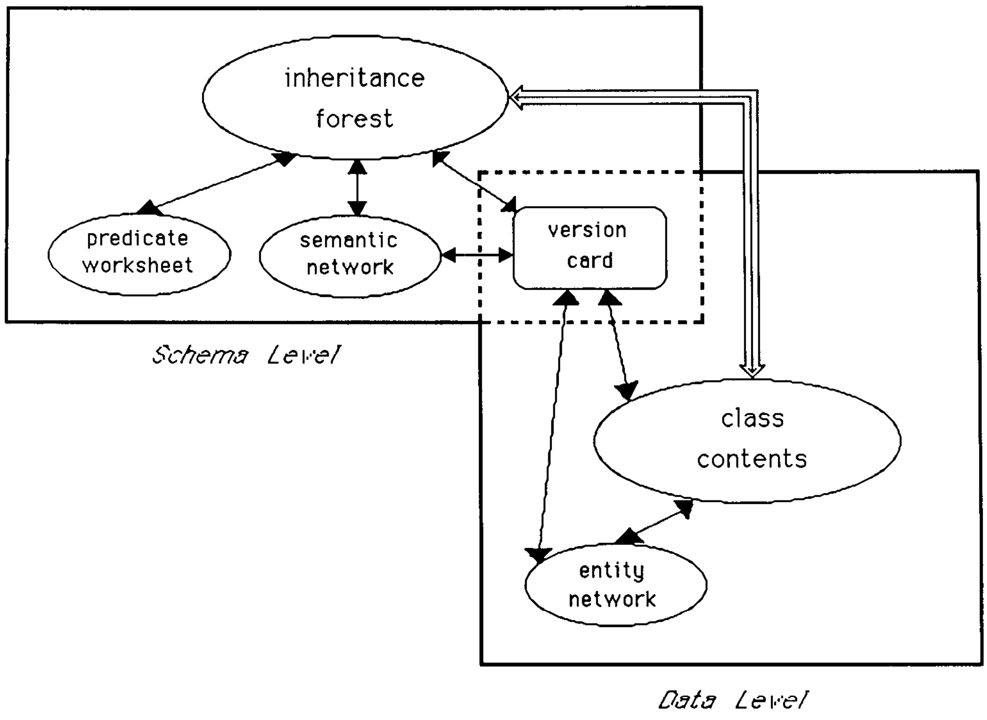

Fig. 1. Interconnections of ISIS schema and data levels.

of the class to signify that it has succeeding versions. Only when the user is viewing the most recent version of the class does the succession indicator arrow disappear.

If the hand icon is not pointing to any schema object, the user is working with the database in its entirety. Toggling to the version menu keys at this point will cause the transaction menu to appear. It is now possible to complete transactions, causing a new version of the entire database, or to view preceding and succeeding transactions. As a final note, the user must be in this screen in order to exit ISIS.

The semantic network provides an alternate view of the currently selected database object (if the current object is an attribute, then the current selection will automatically change to the attribute's value class before entering the semantic network) (see Figure 6 p. 245). The semantic network depicts two types of mappings. The right side of the screen names each attribute of the current class and draws its associated value class. The left side of the screen shows those classes that have an attribute having the currently selected class as its value class. Navigation occurs by picking any class box that maps into or out of the current class. This class then becomes the current class, and the semantic network is redrawn. Upon return to the inheritance forest, the hand icon will point to the new current class, unless the new current class is predefined, in which case there is no current object. The user can also toggle the menu keys to the version menu in this screen, and again, all objects which have past versions will be redrawn with the three-dimensional outline, and all objects with succeeding versions will have the succession indicator arrow at the lower left corner.

The predicate work sheet allows for the construction of derived subclasses and derived attributes. The predicate for a derived subclass S is used to indicate which members of parent(S) are also members of S. A derived attribute is computed from other class and attribute information in the database. ISIS views the definition of a newly derived attribute (or class) to be equivalent to constructing a query. The query is formed by applying attributes as functions to sets of objects to map those objects to other sets of objects. A mapping is depicted as a stack of class icons, one icon for each attribute application. A query clause is represented by two stacks of classes (i.e., mappings) between which a chosen comparison operator is placed [6]. The query is composed of clauses in disjunctive or conjunctive form. The result of the query is saved in the derived class for further examination. These clauses are constructed by editing atoms using the mapping facilities provided to specify the two operands and the operator. No version control is currently provided in this screen.

2.3.2 *Data Level.* The data level is used for browsing the current state of the database and for updating the database contents. The data level provides a view of the data plane through the use of two different screens: *class content* and *entity network*. The class content screen can only be entered from the schema level through the inheritance forest, and the entity network can only be entered from the data level through the class content screen.

When first drawn, the class content screen displays a page containing the current schema selection, and a pannable list of that class's membership set (see Figure 11, p. 250). If the current object was an attribute, then the new current object becomes the attribute's value class. It is possible to create and delete entities from the current object's membership set, or to select (by highlighting in boldface type) certain entities for attribute reassigning. Navigation is possible at this level by following attributes in the current class. The followed attribute's value class will then become the current class, and a new page containing entities of the value class will be displayed on top of the previous page. Entities highlighted in boldface type on the current top page are attribute values for the set of entities highlighted on the previous top page. Furthermore, the ISIS user can specify a user-defined membership set for a new subclass by selecting members in the parent class, then temporarily entering the inheritance forest to position and name the new subclass.

The class content (i.e., the set of all objects belonging to that class) is treated as a database object and, therefore, can be tracked with the version mechanism. As in the schema level, the user can toggle the menu keys to the version menu and create a new version of the class content, or view preceding and succeeding versions of the class content for the current class. When traversing object versions, each page in the class content screen is redrawn to reflect any changes that were indirectly made to the contents of that page. The box outline and indicator arrow are present on this screen in the same manner as in the schema level, but now indicate the presence of preceding and succeeding versions of the class content for the current class page.

The entity network provides a more isolated view of an entity and its attribute values (see Figure 14, p. 253). This screen is obtained by selecting an entity for viewing from the top page of the class content level. The entity network is similar in structure and usage to the semantic network, and is also illustrated through two types of mappings. The right side of the screen shows the entity's attribute values, and the associated value class for the particular attribute. On the left side of the screen are those classes with member entities which use the currently selected entity as an attribute value. Any class box except the father class box may be picked as the current class under consideration. The entity list for that class may then be panned, which may be necessary, since space limitations severely restrict the number of entities that may be shown for those value classes of multivalued attributes. As in the semantic network, navigation is also possible here by selecting an entity from the currently selected class, whereby the entity network is redrawn with the new current entity and its pertinent information. As navigation occurs, a class content page is added in the background on the class content screen to reflect the navigational changes. The version menu is again available for use in viewing preceding and succeeding versions of the current entity. To simplify the user interface, a new version of the entity is created each time its attribute values are reassigned, and so there is no version creation menu key provided in the version menu. The box outline and indicator arrow are once again used on the father class to indicate that the current entity being viewed has preceding and succeeding versions.

2.3.3 *Version Card.* The version card screen makes it possible for the user to compare several versions of the currently selected database object. This screen may be entered from the version menu in the inheritance forest screen (to view class type versions, see Figure 8, p. 247), the class content screen (to view class content versions), or the entity network screen (to view entity versions, see Figure 15, p. 254). Object versions are drawn on version cards, which are stacked on the left side of the workstation screen. Selecting a version card moves the hand icon, making that version the current one under consideration. The current version card may be moved to the right side of the screen to allow for an expanded comparison of object versions, and pages may also be removed from the right side of the screen and placed back in the stack in the original order. If the current object is a class type or a class content, then the user may follow an attribute to another class by picking that attribute on the current version card. If the current object is an entity, then the user may inspect the attribute values for a single attribute (due to space limitations) by picking the desired attribute. It is then possible to navigate to another entity by choosing that entity from the list of attribute values displayed for the chosen attribute. The now familiar succeeding version indicator arrow, as well as a preceding version indicator arrow, will appear below the top page and above the bottom page, respectively, to announce the existence of succeeding and preceding version cards for the current object. This is necessary because only four stacked version cards will be drawn at any one time. When the arrow(s) appear, the user is able to view the indicated versions by panning to the front or back versions through the graphical editing menu. In such a manner, the user can pan all the versions of an object and move to the right side of the screen those versions which are necessary for the desired

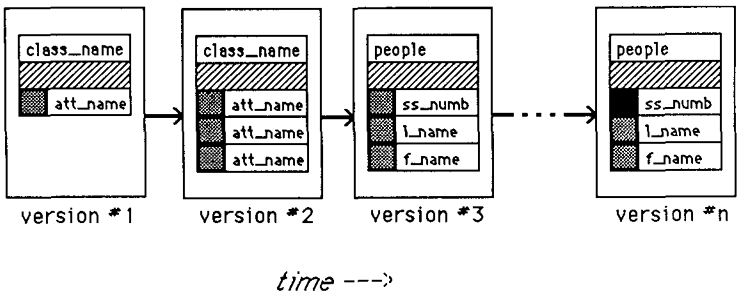

Fig. 2. Linear version set for the class type *people.*

comparison. Upon exit of this screen, the database will remain at the exact state it was in when the creation of the object version on the current version card occurred.

3. VERSION MANAGEMENT AND CONTROL

The following sections describe in detail the primary focus of this extension to ISIS, namely, a system that provides a visual interface to version control in a semantic database model.

3.1 Linear Version Sets

ISIS collects new versions of an object in the object's version set. The version set represents an ordered collection of version instances for the object in question. Each time the user creates a new version of the currently selected object, a new instance of the object is placed in the version set for that object.

This manner of collecting versions coincides with the notion of a linear version set [19]. The linear version set produces a version collection as described above, where each version instance, except the oldest, has a unique predecessor, and each version instance, except the most recent, also has a unique successor. A new version is always added to the version set as the successor of the latest version. If the object has been altered in some way since its most recent version has been created, it is assumed to be in the middle of the next version creation. Figure 2 shows some of the members of a possible linear version set for the class *people.* The arrow between the version instances represents the passage of time between the creation of object versions.

3.2 Transactions

When the database user operates on a database, he or she performs a series of operations that affect the objects in the database. Often these operations are logically related. For example, when new employee Jane Q. Worker begins employment at the Widget Corporation, all the information that the Widget Corporation deems important about Jane will have to be entered into their company database. When this occurs, it is often desirable to group the operations together such that their cumulative effect is guaranteed to be atomic. ISIS provides such a capability and refers to the ordered set of operations as a *transaction.*

As the user modifies objects in the database, ISIS places copies of these objects into an open transaction. The user can close the transaction by using the *complete_transaction* operation described below, or by exiting ISIS, which automatically closes and saves the current transaction.

Transactions themselves are also placed into an ordered grouping—the linear version set for the entire database. The database object is simply the set of all known objects and can evolve like any other object. This allows ISIS to record information about the relative occurrences of transactions. Thus the linear version set for the database incorporates information about all the operations ever performed on the database. This makes it possible for the user to view preceding and succeeding database transactions, using operations described below, to browse through a series of alterations made to the database.

A new ISIS transaction begins as soon as the previous transaction is completed (or when the user first starts ISIS). As objects are modified, the transaction receives its own "copy" of the object, signifying that some change was made to the object in this transaction.

Figure 3 gives a graphical demonstration of the creation of a series of transactions, showing how the completion of each transaction provides a snapshot of the database changes. When transaction 1 ends, it has a copy of two classes, since each apparently was created in this transaction. When transaction 2 opens, we see outlines of the two classes, since neither has yet been modified. As transaction 2 progresses, an attribute is added to one of the classes, and so a new copy of the class is brought forward from the first transaction. Since transaction 2 closes at this point, a new copy of the other class is never obtained. Transaction 3 makes several changes to both classes, so a new copy of the left class is brought forward from the second transaction, and a new copy of the right class is brought forward from the first transaction.

3.2.1 *Transaction Operations.* The following transaction commands are accessible while in the inheritance forest with no currently selected object (i.e., the entire database is the current object), with the version menu toggled on:

complete_transaction (transaction_id)	$\rightarrow$ transaction_version_set
delete_transaction (transaction_id)	$\rightarrow$ transaction_version_set
preceding_transaction (transaction_version_set)	$\rightarrow$ transaction_version_set
succeeding_transaction (transaction_version_set)	$\rightarrow$ transaction_version_set
latest_transaction (transaction_version_set)	$\rightarrow$ transaction_version_set

The *complete_transaction* operation terminates the current transaction and begins a new one. It is analogous to *commit* in conventional transaction processing systems. It does this by making new versions for any objects that were modified by the transaction and adding a new database object to the database version set. Transactions are atomic units of work. The results of a transaction T are not made available to other concurrent transactions until T is explicitly ended.

The *delete_transaction* operation allows the user to delete the current transaction. ISIS automatically closes the transaction and then deletes it. Of course,

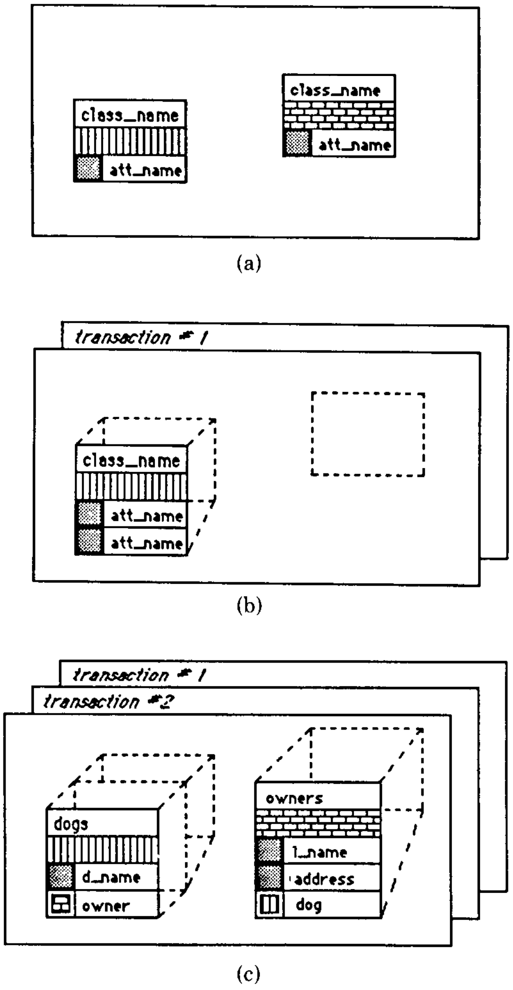

Fig. 3. Three ISIS transactions: (a) Transation 1; (b) transaction #2; (c) transaction #3.

it is possible to be in a new transaction that has made no modifications. In this case, ISIS deletes the most recently completed transaction. In the process of transaction deletion, all operations that comprise the transaction are "undone" (i.e., the effect of the command on the database is removed).

The *preceding_transaction* operation traverses the database version set backward to obtain the database state that precedes the current one. If the current transaction is the first one completed (the "oldest"), then an error message occurs. Otherwise, the operations in the current transaction are simply undone, leaving the database in the same state that it was in when the new current transaction was originally completed. Once again, if the user is in the middle of creating a new transaction, then this transaction is automatically, but temporarily, closed and all its member operations are undone. When the user is finished traversing the transaction versions, and uses the *succeeding_transaction* or *latest_transaction* commands (described next) to get back to the most up-to-date version of the database, then this temporary closing of the current transaction will be undone, leaving the user in precisely the same state as before the traversal began.

The *succeeding_transaction* command provides an inverse operation to *preceding_transaction*. ISIS is placed in the state that it was in after the current transaction completed. The transaction that was in force at that time is made current. If the current transaction is the most recent, then an error message is given. Otherwise, the operations that comprise this transaction are redone to make the database consistent with its state when that transaction was first completed. If the use of this operation yields the most recent database version, then ISIS checks if the current transaction was temporarily closed. If this is true, then it is reopened for additional database changes.

The *latest_transaction* command produces the most recent version of the database by successively calling *succeeding_transaction* until no more transactions can be "redone." This command is used by ISIS when toggling back to the standard menu, since the standard menu must always be accompanied by an up-to-date database (database altering operations cannot be performed on objects if the database is not current).

3.2.2 *Transaction Representation.* An ISIS transaction must store information regarding each of its operations so that each can properly be un/redone when performing version traversal. When a new transaction is completed, we cannot afford to create a new instance of the entire database to reflect the changes made in the most recently completed transaction. Instead, we need to store only the pertinent operations, or the database *deltas*, for each transaction.

As an ISIS session progresses, operations that affect the state of database objects are placed on a system undo stack. When the user wishes to complete the transaction, the undo stack is examined. If the stack is not empty, then a new transaction entity is added to the TRANSACTION class. The information in the undo stack is popped off and placed into the entity, preserving the order so that transaction traversal will affect objects in the proper order.

Each transaction, then, is represented as a uniquely identified entity in the TRANSACTION class. This entity has a series of attribute values that describe any information necessary to un/redo transaction operations. For example, there is an attribute that contains a list of the operation codes for the commands in the transaction and an attribute that contains a list of object ids of the objects affected by the operations.

3.3 Object Versions

ISIS supplies version control mechanisms for three types of database objects: class type, class content, and individual entities.

The ISIS user can save versions of class types as they are altered during creation time. It is then possible to view the evolution of the class type from its initial conception to its present state. It should be noted that, since attributes merely provide descriptive information about the class, they have no version sets.

However, changes performed on the definition of a class's attributes make the creation of a new class version possible.

A class content version for the current class may be created while in the class content level of ISIS, with the version menu toggled. This is useful as the database data content changes and evolves. For example, the user may wish to create a new instance of the class *employees* to represent the addition of several new workers to the current set of company employees. The presence of at least one *create_entity* or *delete_entity* operation that affects the current class is sufficient to enable the user to create a new version of the current class contents.

A new entity version is created each time the ISIS user reassigns the entity's attribute values. (For the sake of simplicity in the interface, users need not be aware of version control of entities. It happens automatically as changes are made.)

3.3.1 *Version Operations.* ISIS is capable of recording a version history for all objects in the system including class definitions and the database itself. New versions are created automatically for instances whenever an attribute value is changed and for classes at the end of a transaction. ISIS allows the following operations on version sets:

```
preceding_version (object_id, version_id, trans_id)  → version_set
succeeding_version (object_id, version_id, trans_id) → version_set
latest_version (object_id, version_id, trans_id)     → version_set
view_versions (object_id)                            → version_set
```

The *preceding_version* operation returns the database to the state that it was in after the transaction that created the previous version. ISIS considers the database user always to be in the middle of a version creation for each object. Thus, if the database is in its most recent state when this operation occurs, ISIS checks to see whether it is possible to create a new version of the current object. If the creation can be performed, then the new version is temporarily created. The preceding version can then be obtained by finding the version that precedes the newly created one. If the database is not in its most recent state, or if it is not possible to create a new object version, then ISIS simply finds the version that precedes the current object version. If the current version is the first one, then an error message is given. Operations are undone in previous transactions (even those that do not affect the current object—to keep the database consistent) until ISIS finds the previous version for the current object. Upon returning to the database's most current state, ISIS undoes the temporary version creation, leaving the database in exactly the state that it was in before the traversal began.

As in transaction handling, the *succeeding_version* command provides an inverse operation to *preceding_version.* The object's version set is traversed until the version that succeeds the current version is obtained. An error message is given if the current version is the most recent one. Otherwise, transaction operations are redone until the succeeding version creation is reached. This places the database in a state that is consistent with the state of the database when that version creation was first done. If the result of the operation yields the most recent object version (which is always the most recent database version), then ISIS checks if the current object had a version temporarily created. If this happened, then it is uncreated, and the database is once again consistent.

The *latest_version* command calls the *succeeding_version* command until the database reaches its most recent state. ISIS uses this operation to obtain an up-to-date version of the database before toggling back to the standard menu to perform new database operations. For the user, this command provides a fast way to obtain a representation of the most recent version of a database object.

The version card screen is accessed from the inheritance forest, class content, and entity network screens through the *view_versions* command. This screen makes use of the *preceding_version* and *succeeding_version* operations to traverse the database to find and draw various versions of the current object.

4. EXAMPLE: AN OFFICE APPLICATION

The following two-part example illustrates the primary features of the ISIS visual interface to version management. The first part of the session demonstrates the use of transactions and object versions in the initial schema design and data entry. The second part of the example shows the use of object versions as database objects evolve over a period of time.

4.1 The Application Environment

The database designer for a typical office has decided that the office environment can be described using five base classes, one grouping class, and one subclass. The *people* base class has four attributes: *last_name*, which identifies the person entity; *jobs*, a multivalued attribute mapping into the *jobs* class, which lists the jobs performed by the person; *department*, which maps into the *departments* class and gives the person's department; and *supervisor*, which maps back into the *people* class and gives the supervisor's name. The grouping *by_supervisor* groups company employees according to their supervisors.

The *jobs* base class has two attributes: *job_name*, which identifies a job by title, and *classific*, which classifies the job as clerical, technical, or managerial. The subclass *secure_jobs* of the base class *jobs* is a user-defined subclass which has members that require the job performer to first obtain special security clearance. The *clear_level* attribute maps into the INTEGER base class and gives the level of security clearance necessary for the job.

The *reports* base class has three attributes: *report_name*, which identifies the report by title, and *producers* and *receivers*, two multivalued attributes mapping into the *people* base class, which list the people who produce and receive the report, respectively.

The *chapters* base class consists of three attributes: *chapter_name*, which identifies the chapter by name; *chapter_num*, which maps into the INTEGER base class and gives the chapter number; and *report*, which maps into the *reports* base class and indicates to which report the chapter belongs.

The *departments* base class consists of two attributes: *dept_name*, which identifies the department by name, and *interactions*, a multivalued attribute that maps back to the *departments* base class and lists those departments that directly interact with a particular department.

4.2 Sample Database Setup

The database designer begins with an empty database which has been titled *office.db*. In the first database transaction, the designer creates the base classes that are explained above. Each base class also received a new object version after the class and identifying attribute were properly named. As the user modifies the base classes set up in the first transaction, he creates a new version of each. Figure 4 shows the user as he is about to create a new object version of the class type *chapters*. Other base classes in the database already have a boxed outline representing the fact that they have at least one previous version; after the user finishes this operation, the currently selected class will also have the version outline. Figure 5 depicts the user completing the final transaction for the schema design.

As an added measure of caution, the designer decides to review his creation process, in case something was overlooked. He selects the *reports* base class, and enters the semantic network screen in Figure 6. Examining the screen, he can see that the base classes *reports*, *people*, and *chapters* all have past versions, which he realizes is correct, since he created several object versions of the classes as he built the schema. For comparison purposes, he decides to traverse the version set for the *reports* class, to get its preceding type version. The result is shown in Figure 7. Here he notices that the *reports* and *people* classes now each have preceding and succeeding versions, and that the *reports* attribute in the *chapters* class no longer maps into the current class. He realizes that this is because value class selection for that attribute was made after the creation of this particular instance of the *reports* base class.

Wishing to further examine the changes made between the various version creations for the *reports* class, he proceeds to the version card screen of ISIS (Figure 8), since this screen allows for many of the linear version set members to be viewed at once. There he sees four stacked version cards for the class type, each card representing a created version instance. Figure 9 shows how he has popped one of the version cards to the top of the stack and moved the most recent version card out to the right side of the screen, so that the two class type instances can be compared more easily. Noticing the arrow which signifies that there are more preceding versions than those represented on the screen, he decides to pan to those preceding versions in Figure 10. Since the currently selected card was still sitting in the unpopped stack of version cards (see Figure 9), ISIS automatically moves the card to a designated spot on the right side of the screen, and then draws the only other preceding version for the *reports* class type.

Satisfied with the present schema setup, the designer proceeds to the data level and enters the company's pertinent data (as entities) for its current operating environment.

4.3 Using ISIS as the Office Evolves

In this portion of the scenario, we rejoin our user at a later point in time. The company has expanded somewhat, and wishes to add a new marketing department to the existing research/development, production, and sales departments. This new marketing department will consist of two new employees, Walter, as department head, and Preston, as his marketing staff. The marketing department will not produce its own report, but instead Preston will produce a chapter in the research report (based on his infinite marketing wisdom).

Our database user enters the data level with the class content screen and adds the two new employees to the *people* class. Figure 11 shows the two new workers highlighted in the entity list for the class. The user has just created a new version of the *people* class content, to represent the change in the number of employees for the company. The boxed outline appears around the class to indicate the presence of a past version for the *people class*.

Figure 12 shows the user in the process of assigning the new employees to the marketing department. In this figure, the user has just added the marketing department to the *departments* class and has just created a new version of the class content. Again the boxed outline appears to indicate the presence of a past instance of the class content for *departments*. Out of curiosity, the user wishes to see how the class content for the *departments* class looked before this new addition, so he views the preceding class content. Figure 13 shows the result of this operation. The *departments* class shows the existence of only three departments now and no longer has a past class content version, but it does have a succeeding version, which is the one created in Figure 12. The user also notes that the *people* class no longer has the two new marketing department employees highlighted in the entity list, because they were not yet added to the company when this version of the *departments* class content was created.

The user now turns his attention to altering the research report to reflect the addition of new marketing information from the marketing staff. After making the desired changes to the res_report (research report) entity, he travels to the entity network screen, where he sees the most recent version of the research report and its attribute values (Figure 14). The presence of the boxed outline indicates past versions of the entity. The user also notes that the report now has three chapters to it, which map into the res_report entity from the *chapters* class.

The user wishes to see who has received copies of the research report at past times, and so he goes to the version card screen for the res_report entity in Figure 15. Once there, he pulls out the two versions that directly precede the current version. After picking the receivers attribute on the current version card, he sees who has received the report in past versions. He notices that Walter only receives a copy in the most recent version, and this makes sense to him, since Walter just recently joined the company.

Now our user turns his attention to updating the chapters of the research report. The report previously had three chapters, res_intro, res_content, and res_summary. The report should now contain four chapters, res_intro, res_market, res_develop, and res_concl. Res_market and res_develop were originally the single chapter res_content, and res_summary has been renamed to res_concl. After performing all these changes, the user wishes to compare the current chapter content of the research report to previous versions of the report. The user reenters the class content screen with the chapters class as the currently selected object. He picks the res_conclus entity and moves to the version card screen in Figure 16. After moving out a preceding version card, he sees right away that both the res_conclus chapter and the res_summary chapter were chapters in the research report, the res_conclus chapter was at one time called res_summary. He recalls that this is correct, since the chapter was renamed during the updates made when the marketing information was added.

In a final browsing, the user stays in the version card screen, and traverses to the entity res_report, to see who receives it, and then follows again to the res_report receiver, McGuire, to see what his job is. Satisfied with the current state of the database, the user exits the version card level, and returns to the class content screen, where he originally was before the entity traversal began (Figure 17). He notices that, although he left the screen originally with only the class content for *chapters* appearing, the screen now shows two additional class content pages, with entities highlighted. He realizes that this is an indirect result of his entity traversal in the version card level, and that ISIS automatically added the proper pages to reflect the database entity browsing.

5. FUTURE EXTENSIONS

The first area of exploration involves the definition of version-related predicates for subclass and attribute derivation. This would entail a facility which allows the user to form a query that could gather information based on a previous state or several previous states of the database. For example, if the predicate work sheet is used to define the membership set for subclass X at the most recent state of the database, it might be interesting to see what X's membership set would look like if the predicate were evaluated at some previous, noncurrent state of the database. It would also be desirable to design a set of version-related operators, which included a notion of time when being evaluated. For instance, it would be useful to know which versions of the class contents for class C include entity E as a member. At its present state, ISIS provides no version control at the predicate level, other than the fact that committing a predicate is an operation in a transaction.

The second direction explores the use of a more powerful and complicated version set than the linear version set. A branching version set [19] provides for *alternatives* in the version representation of objects. Such a version set allows more than one version be the successor of a given version. Each alternative may evolve as a chain of versions. These chains may branch as well, to form new alternatives. The splitting of a version set at some point into two or more branches represents a situation in which several competing instances of an object must be simultaneously maintained.

REFERENCES

Note: References [1]–[4], [8], [11], [12], [14]–[16], [18], and [20] are not cited in text.

1. BILLER, H., AND NEUHOLD, E. J. Semantics of databases: The semantics of data models. *Inf. Syst. 3* (1978), 11–30.
2. BROWN UNIVERSITY. Brown workstation environment: Programmer's manual, preliminary version 2.0. Dept. of Computer Science, Brown Univ., Providence, R.I.
3. CATELL, R. G. G. An entity-based database user interface. In *Proceedings of the International Conference on Management of Data* (Boston, Mass., May 30–June 1). ACM, New York, 1980.
4. CHEN, P. P. S. The entity-relationship model: Toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar. 1976), 9–36.

5. ECKLUND, E. F., AND PRICE, D. M. Multiple version management of hypothetical databases. Tech. Rep. 84-40-1, Tektronix Laboratories, July 1984.
6. GOLDMAN, K. J., GOLDMAN, S. A., KANELLAKIS, P. C., AND ZDONIK, S. B. ISIS: Interface for a semantic information system. In *Proceedings of the International Conference on Management of Data* (Austin, Tex., May 28-30). ACM, New York, 1985.
7. HAMMER, M., AND MCLEOD, D. Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (Sept. 1981), 351-386.
8. HEROT, C. F. Spatial management of data. *ACM Trans. Database Syst. 5*, 4 (Dec. 1980), 493-514.
9. KATZ, R. H., AND LEHMAN, T. J. Storage structures for versions and alternatives. Tech. Rep., Computer Sciences Dept., Univ. of Wisconsin-Madison.
10. KATZ, R. H., AND LEHMAN, T. J. Database support for versions and alternatives of large design files. Tech. Rep., Computer Sciences Dept., Univ. of Wisconsin-Madison, 1983.
11. KING, R. Sembase: A semantic DBMS. In *Proceedings of the 1st International Workshop on Expert Database Systems* (Kiawah Island, S.C., Oct. 1984).
12. KLAHOLD, P., SCHLAGETER, G., UNLAND, R., AND WILKES, W. A transaction model supporting complex applications in integrated information systems. In *Proceedings of the International Conference on the Management of Data* (Austin, Tex., May 28-30). ACM, New York, 1985.
13. LUM, V., DADAM, P., ERBE, R., GUENAUER, J., PISTOR, P., WALCH, G., WERNER, H., AND WOODFILL, J. Designing DBMS support for the temporal dimension. In *Proceedings of the International Conference on the Management of Data* (Boston, Mass., June 18-24). ACM, New York, 1984.
14. MYLOPOULOS, J., BERNSTEIN, P. A., AND WONG, H. K. T. A language facility for designing database-intensive applications. *ACM Trans. Database Syst. 5*, 2 (June 1980), 185-207.
15. REINER, D., BRODIE, M., BROWN, G., CHILENSKAS, M., FRIEDELL, M., KRAMLICH, D., LEHMAN, J., AND ROSENTHAL, A. A database design and evaluation workbench: Preliminary report. In *Proceedings of the International Conference on Systems Development and Requirements Specification* (Gothenburg, Sweden, Aug. 1984).
16. SMITH, J. M., AND SMITH, D. C. P. Database abstractions: Aggregation. *Commun. ACM 20*, 6 (June 1977), 405-413.
17. THOMAS, I., AND LOERSCHER, J. MOSAIX—A version control and history management system. In *Proceedings of the GTE Workshop on Software Engineering Environments for Programming in the Large* (Harwichport, Mass., June 1985).
18. WONG, H. K. T., AND MYLOPOULOS, J. Two views of data semantics: A survey of data models in artificial intelligence and database management. *INFOR 15*, 3 (1977), 344-382.
19. ZDONIK, S. B. An object management system for office applications. Tech. Rep., Dept. of Computer Science, Brown Univ., Providence, R.I.
20. ZDONIK, S. B. Object management system concepts. In *Proceedings of the 2nd ACM SIGOA Conference on Office Information Systems* (Toronto, Ontario, Canada, June 25-27). ACM, New York, 1984.
21. ZDONIK, S. B. Version management in an object-oriented database. In *IFIP 4.2, Workshop on Advanced Programming Environments* (Trondheim, Norway, June). IFIPS Press, Reston, Va., 1986.

Received February 1986; revised June 1986; accepted July 1986

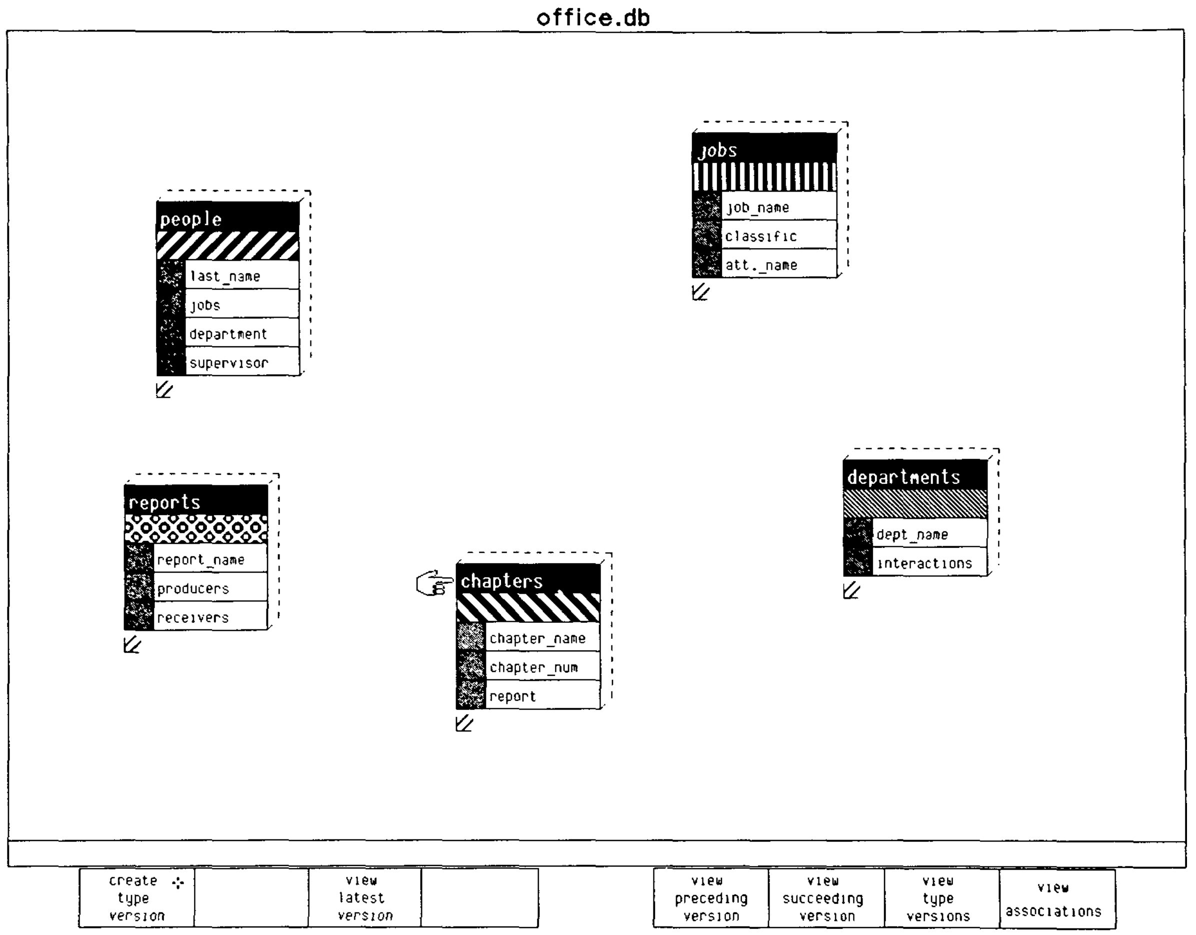

Fig. 4. Before creation of a new version for class *chapters*.

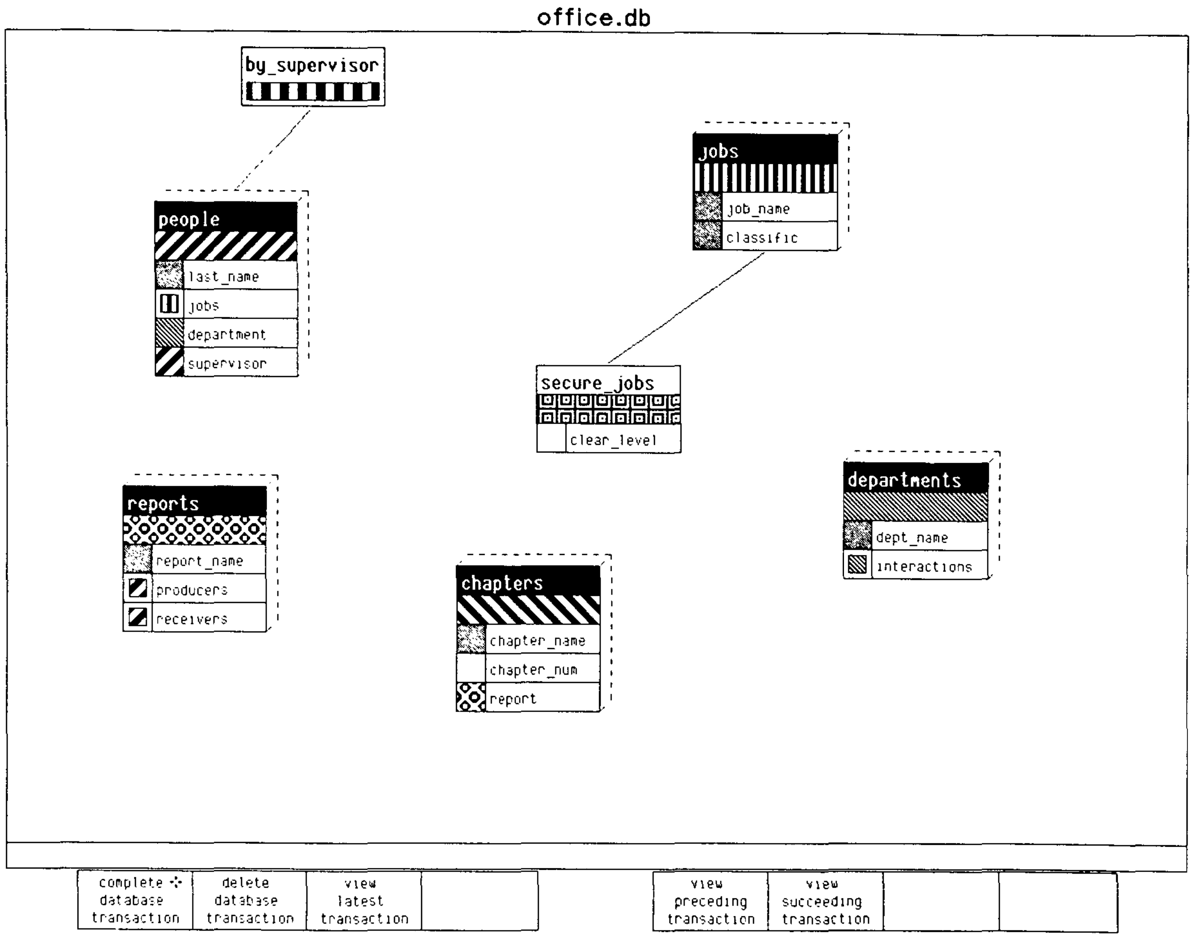

Fig. 5. After completion of the desired schema setup.

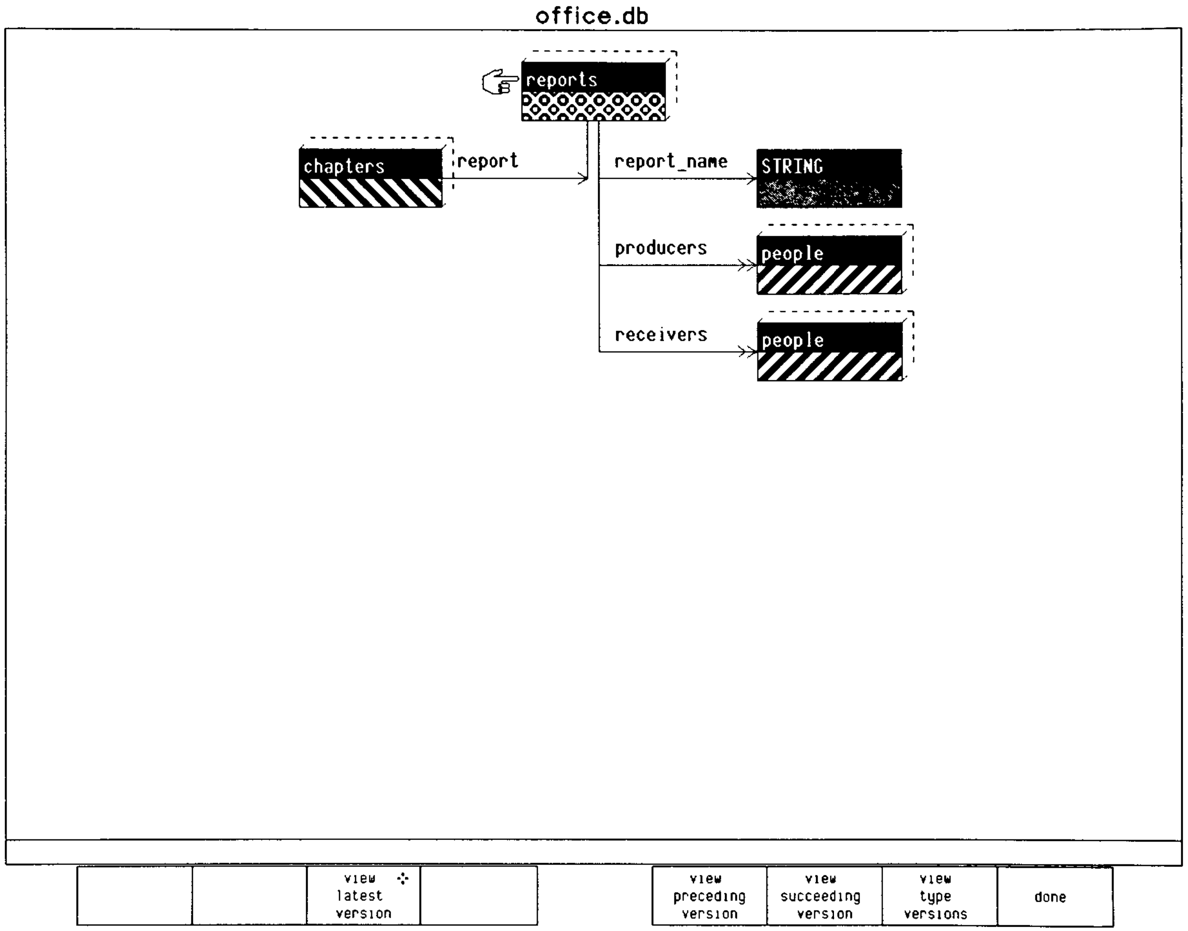

Fig. 6. Most recent class type version of *reports*.

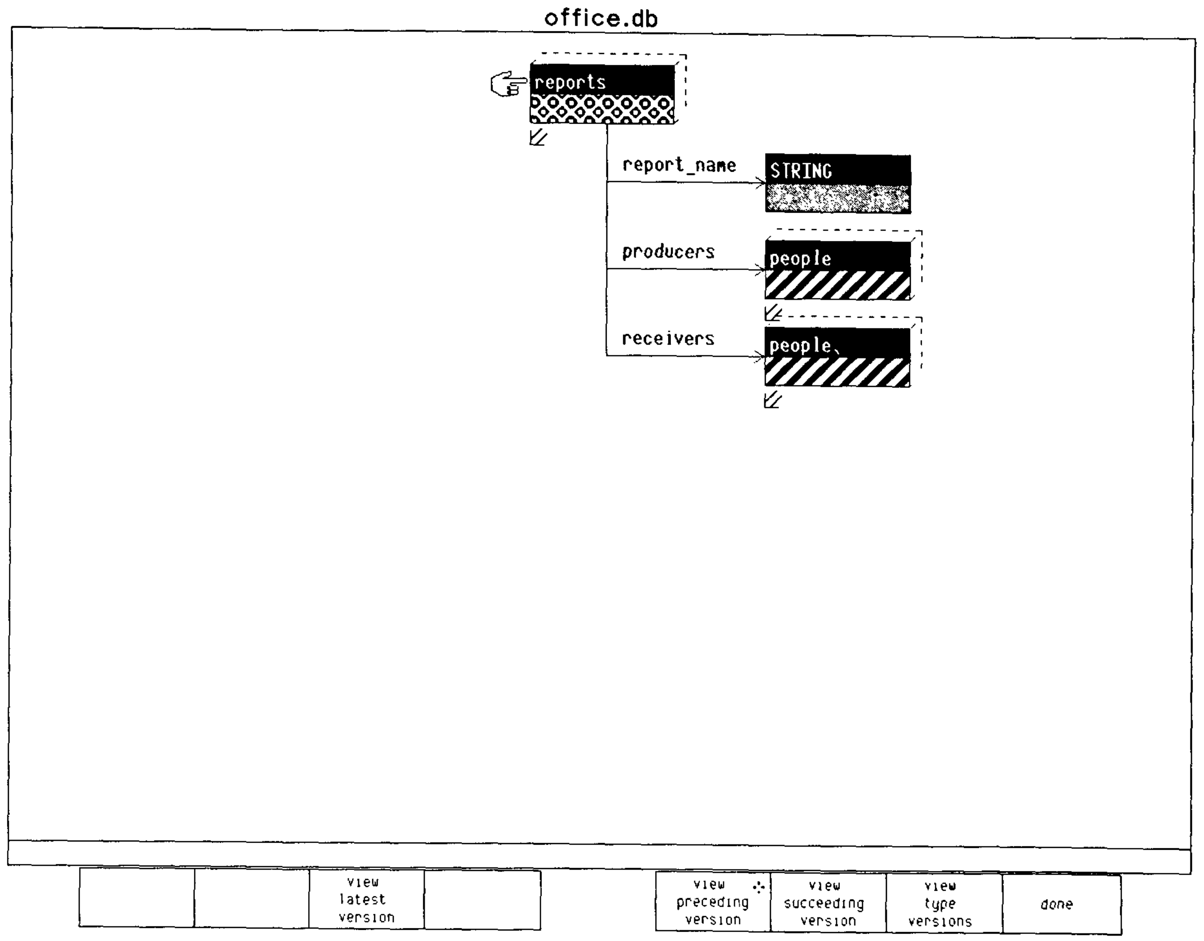

Fig. 7. Preceding class type version of *reports*.

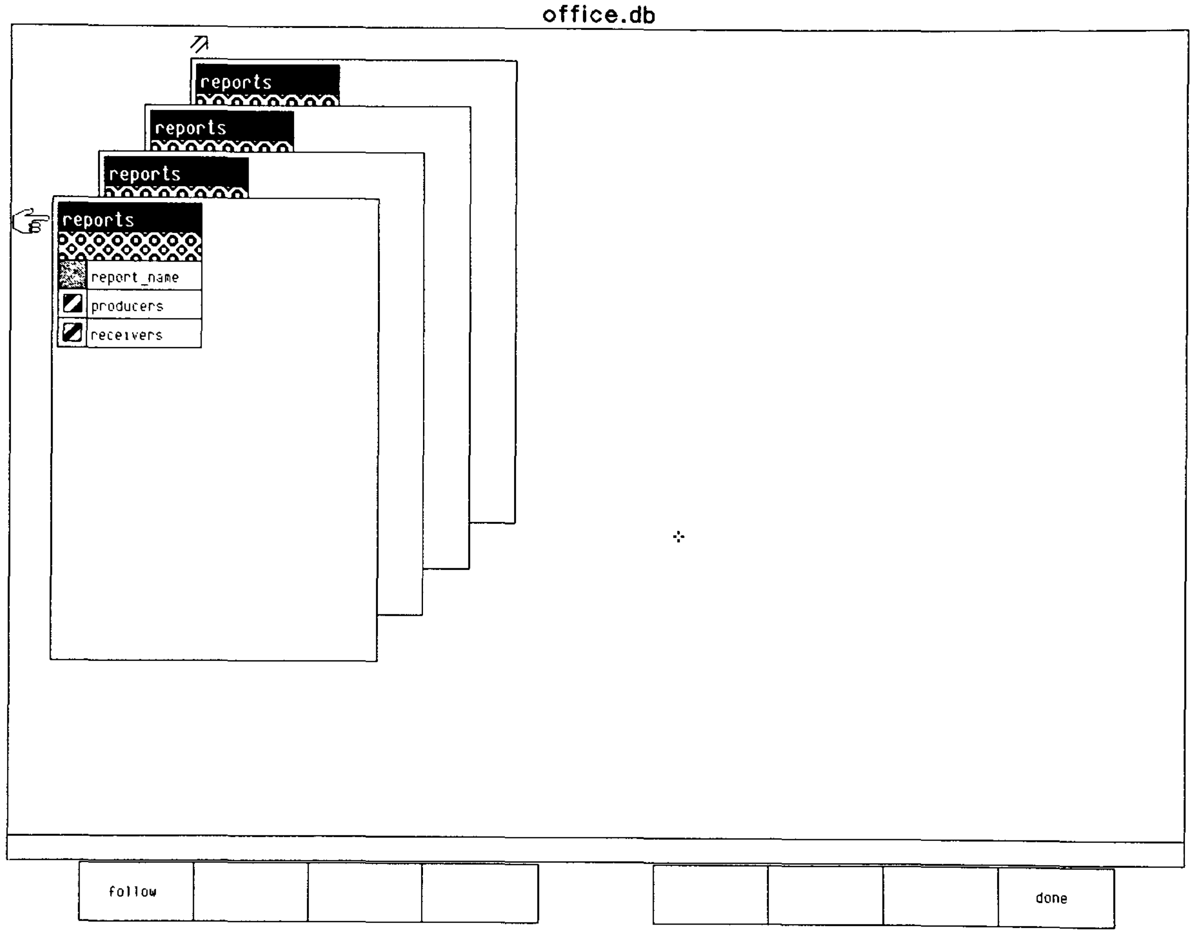

Fig. 8. Unpopped version card set for *reports*.

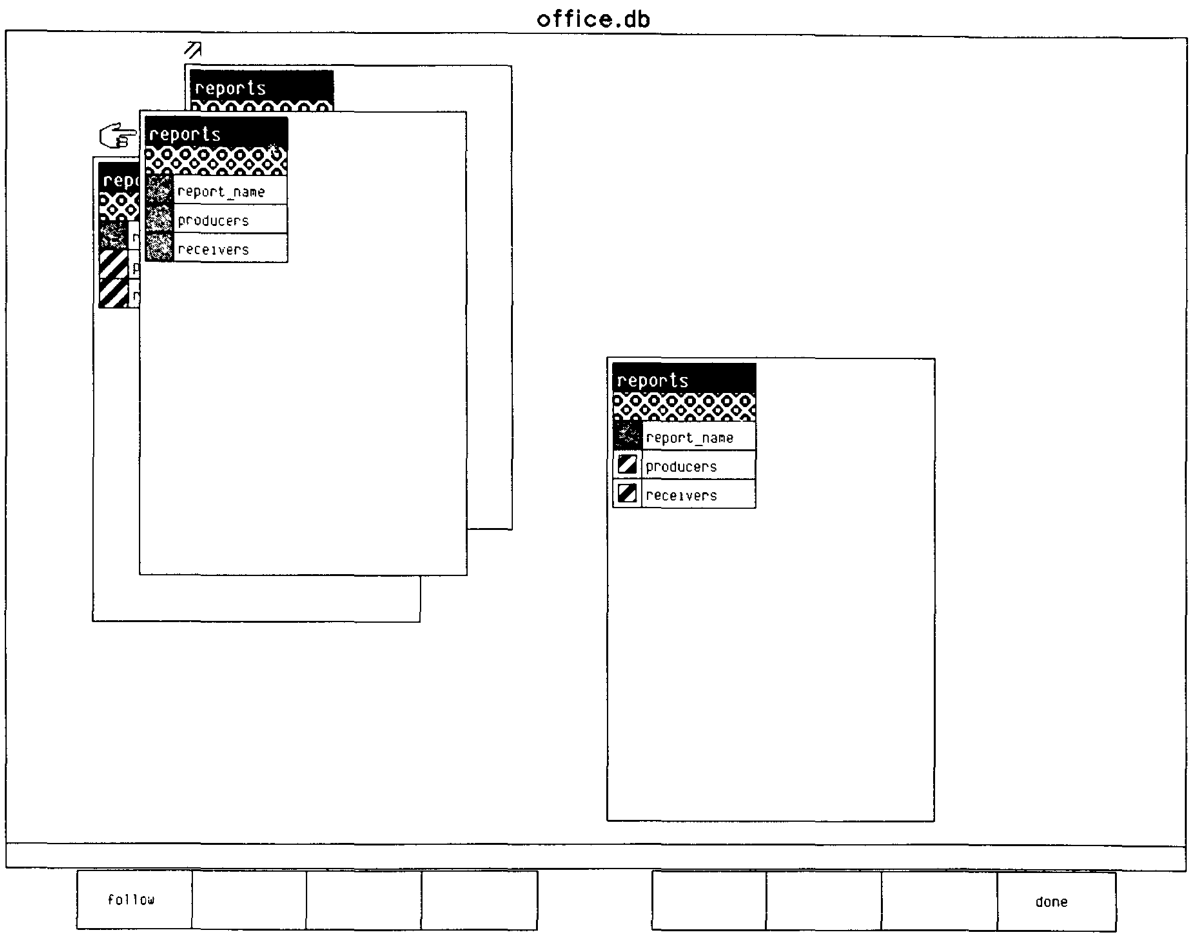

Fig. 9. Comparison of two class type instances.

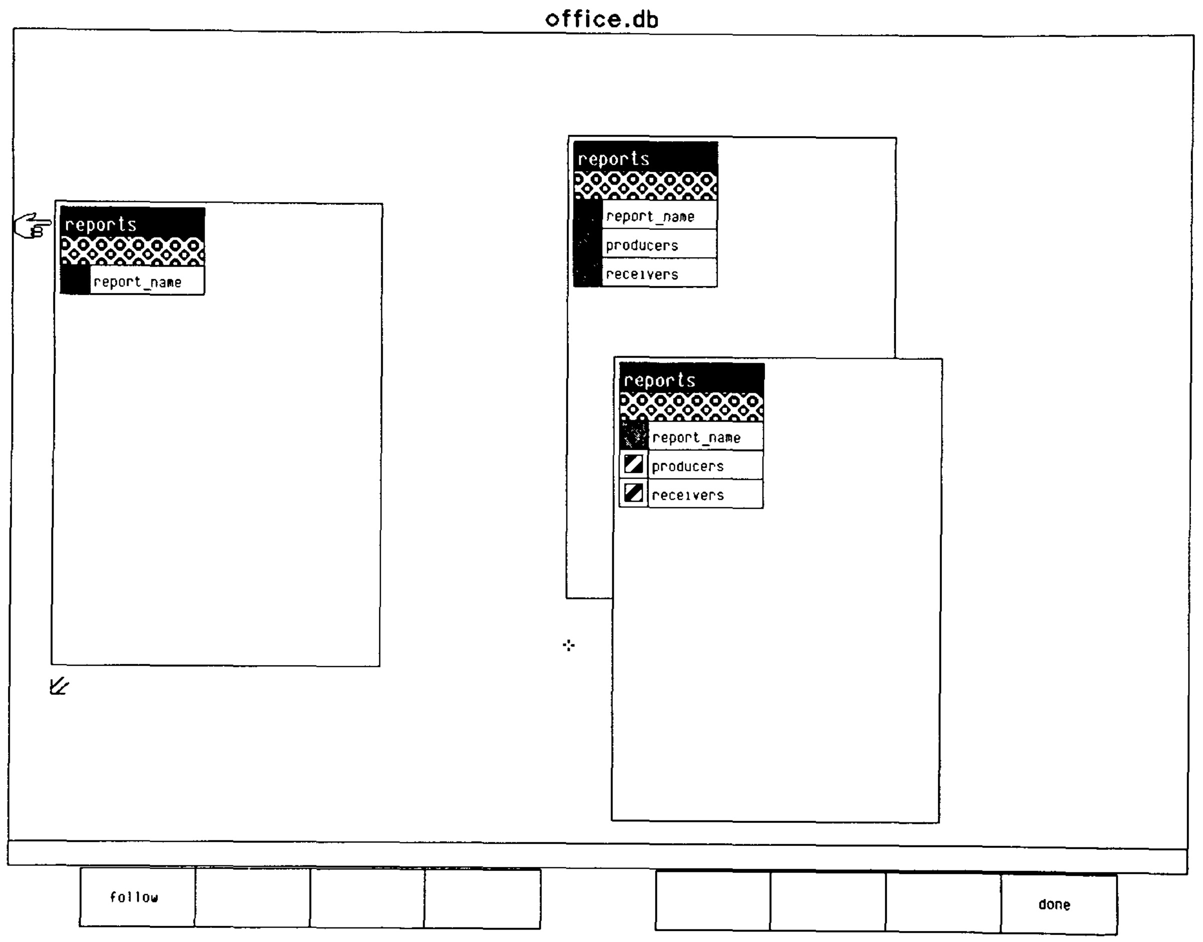

Fig. 10. *Panning the version cards to get previous versions.*

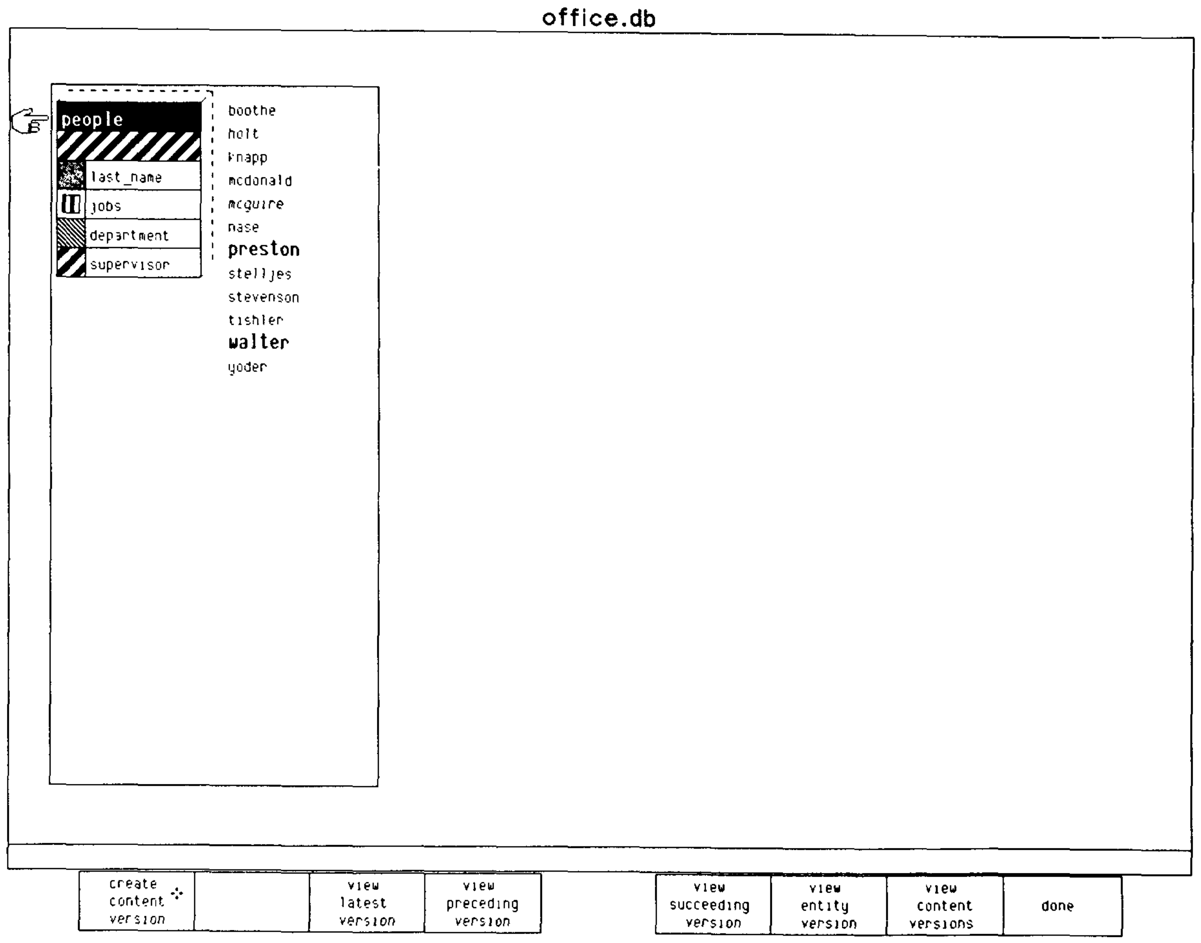

Fig. 11. Creating a new class content version for *people*.

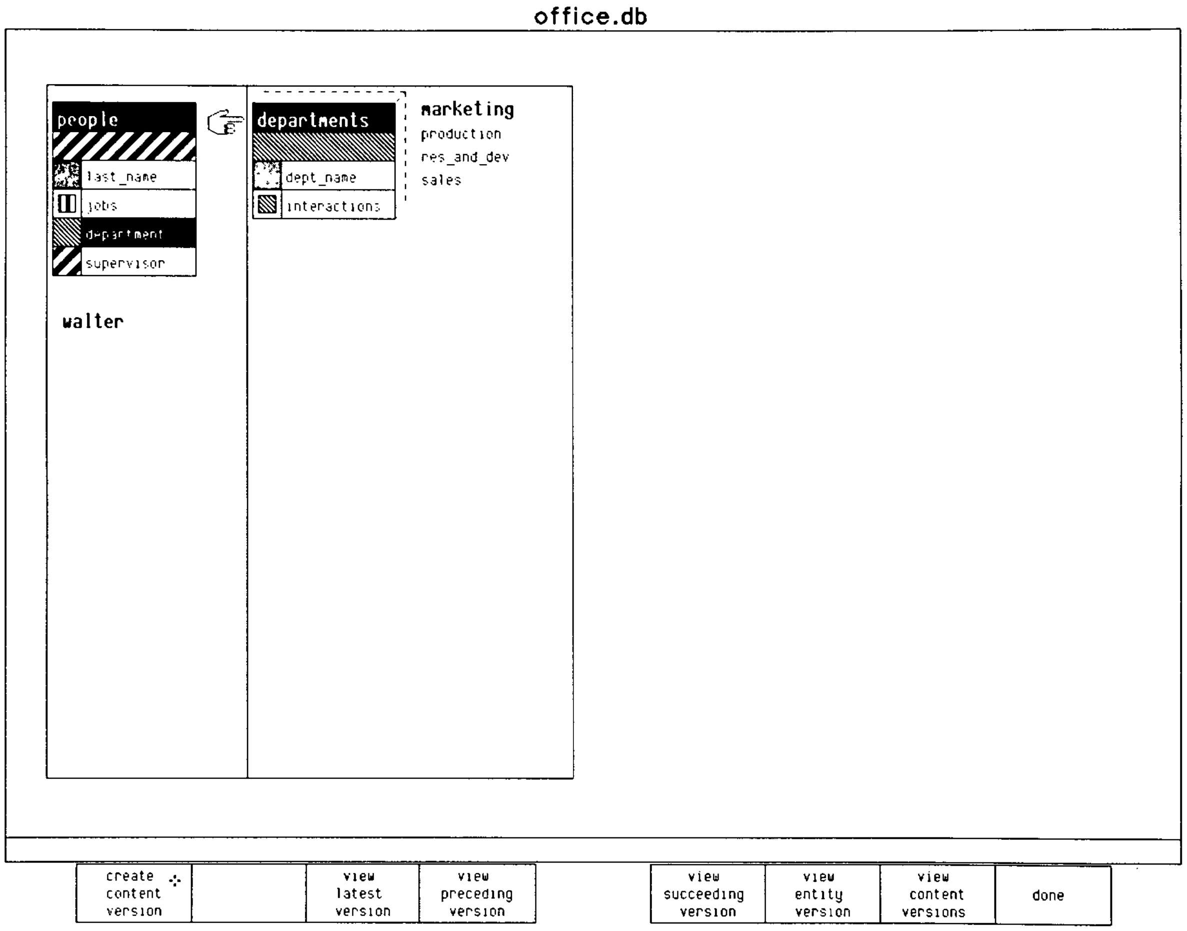

Fig. 12. Creating a marketing department and a new version of the content.

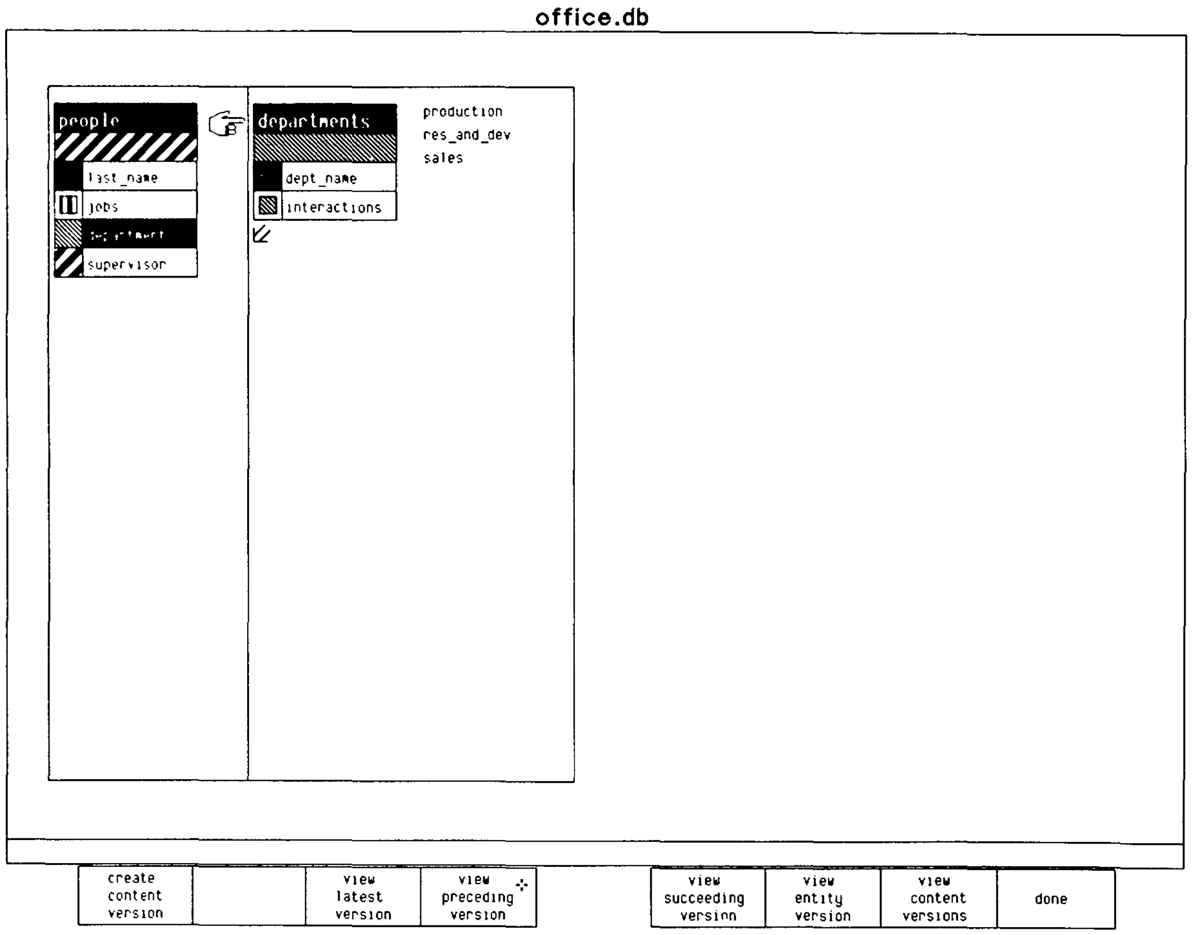

Fig. 13. Viewing a preceding version of the class content for *departments*.

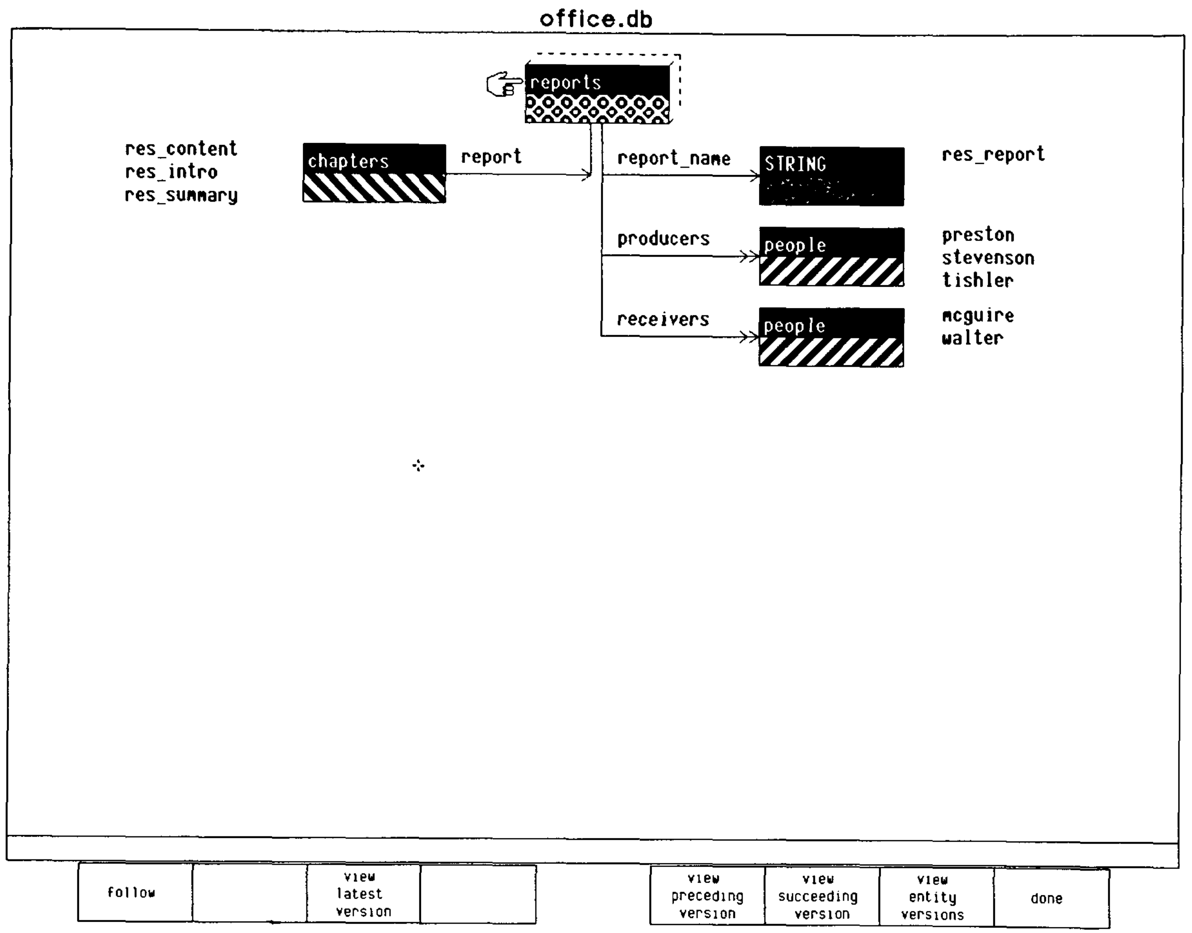

Fig. 14. Entity network for entity res_ report.

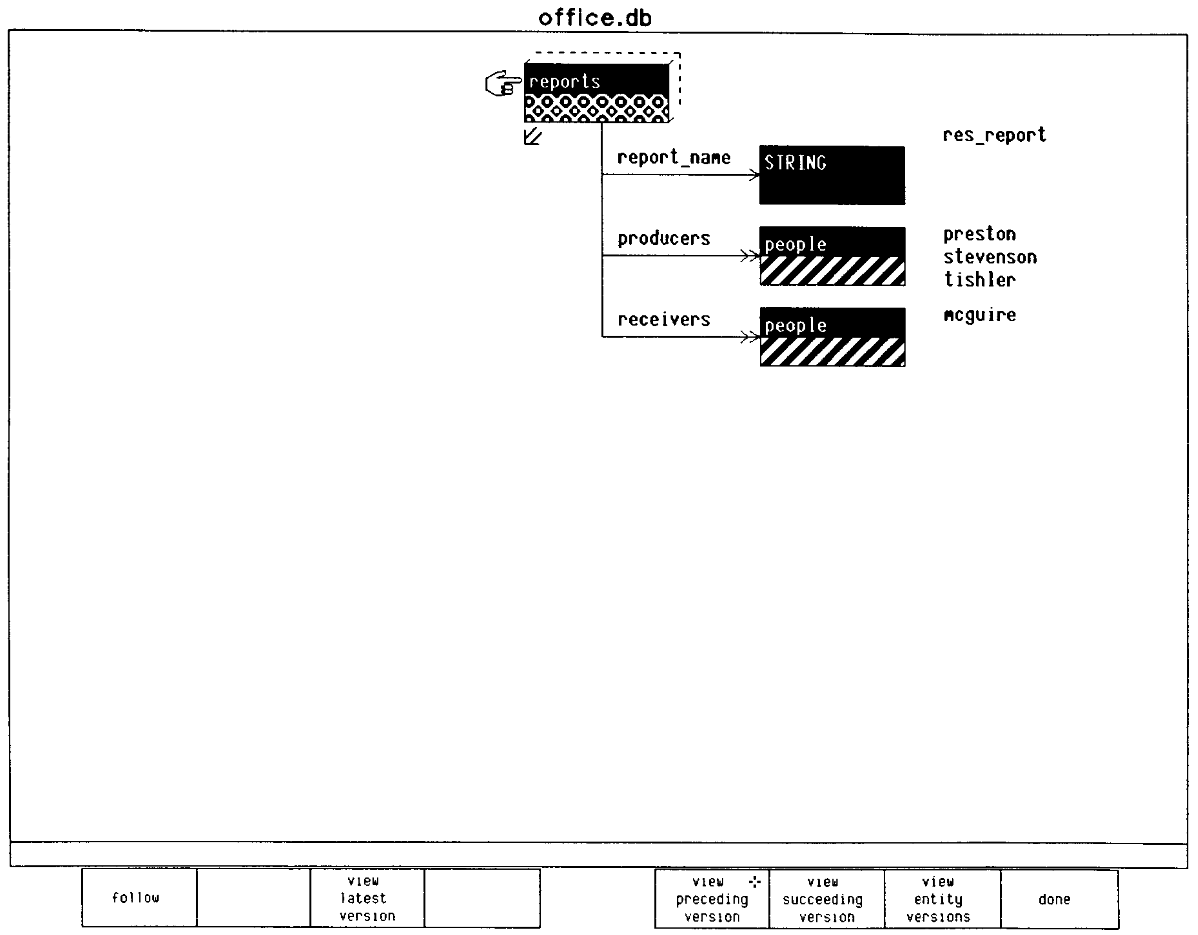

Fig. 15. Viewing a preceding version of the res_report entity.

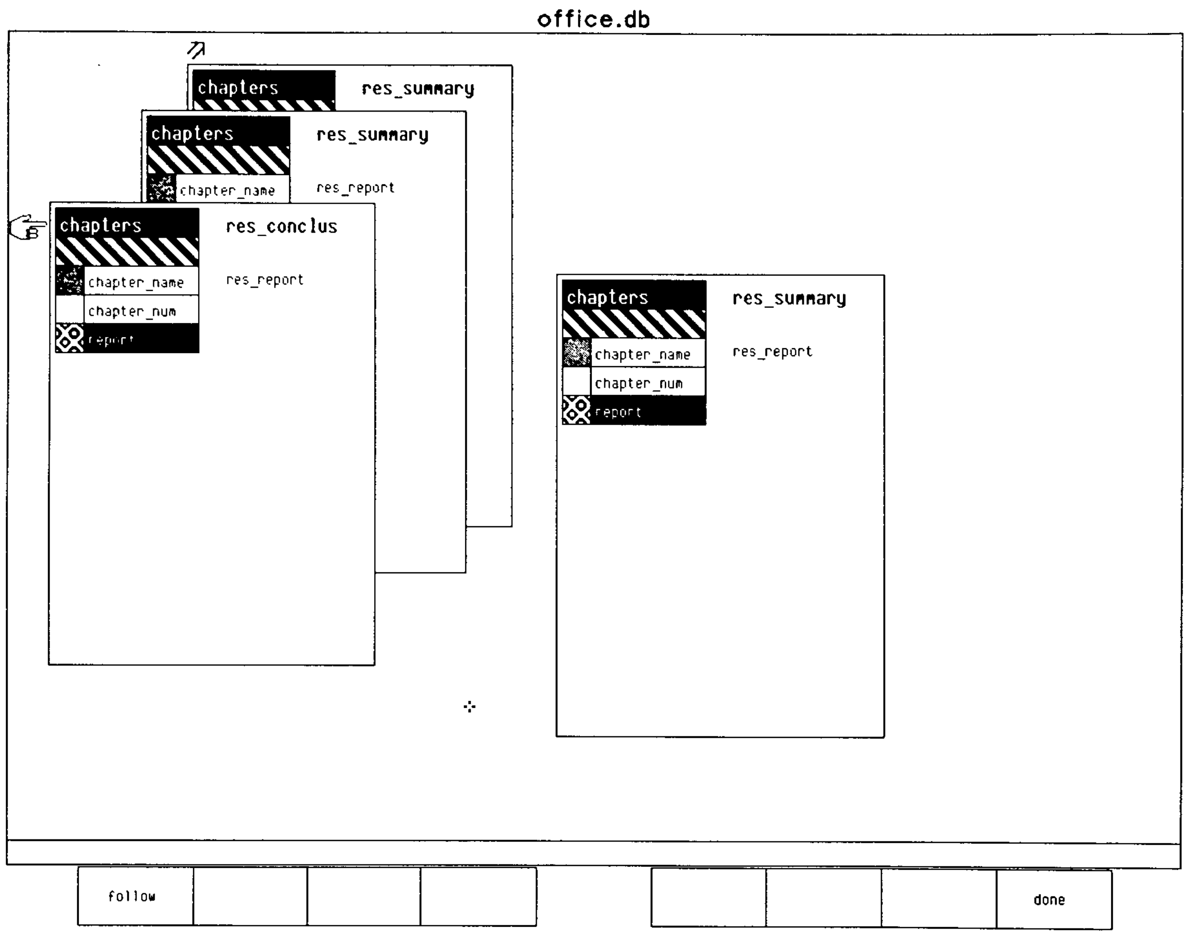

Fig. 16. Viewing a chapter name change in the res_report entity.

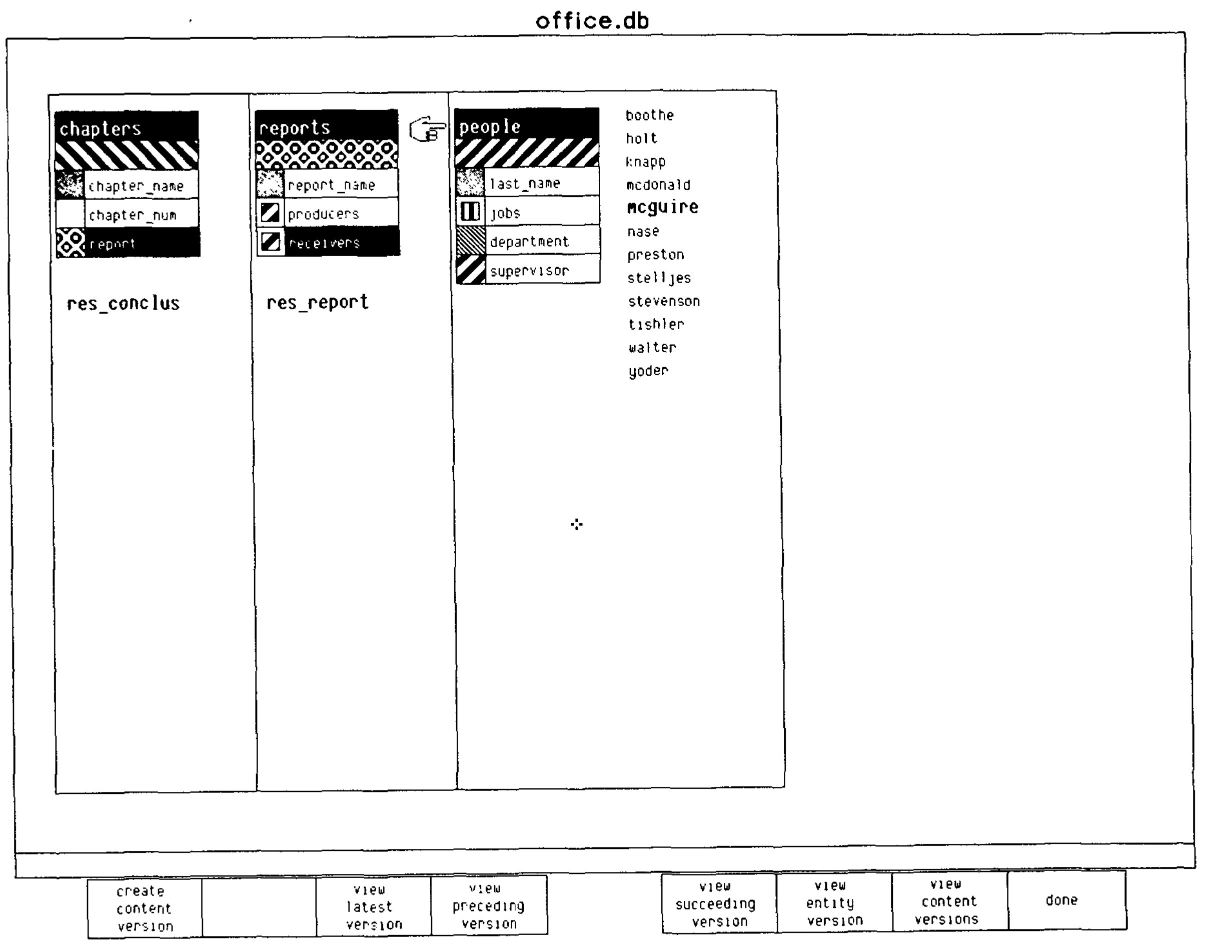

Fig. 17. Reentering the class content screen with new pages present.