A Browser for a Versioned Entity-Relationship Database

Bjørn Gulla*
Norwegian Institute of Technology (NTH), Trondheim, Norway

Presented at the Int’l Workshop on Interfaces to Database Systems (IDS’92)
Glasgow, Scotland, 1–3 July 1992

Abstract

In database systems supporting versioning, there exist two information spaces, the version space and the product space. The version space contains a point for each possible configuration of the product, and the product space consists of the individual components comprising the product. While conventional models represent these spaces in an intertwined structure, the change-oriented versioning model allows them to be treated quite separately.

Existing browsers and navigation tools only support single-version views of the database. However, many tasks require or would be better supported if visualizations also conveying version information were used. A multi-version view is a single pictorial representation of multiple versions of some structure.

We are currently building a browser with a graphical user interface that allow easy navigation in both information spaces for an entity-relationship database. A range of different multi-version viewing techniques is supported.

1 Introduction

For many design databases (for instance, those for software engineering or CAD), the main activity is the design of artifacts. In the early stages of development, multiple alternative designs under parallel development must be handled. Designs are typically combined, refined, merged etc., and structural changes are quite common. Later in the product life-cycle, often platform- or site-specific adaptations or customizations must be made and maintained. In addition, the steady evolution by error corrections and enhancements must be tracked. The need for automatic support of multiple versions in such database system is widely recognized as essential, and an increasing number of databases provide such support [Kat90, EB91].

In the EPOS project we are building a database system based on an entity-relationship model [Che76] extended with uniform versioning and subtyping of both entities and relationships [Lie90]. It can also be considered a structurally object-oriented database [Dit86, ZM90], since new types can be created, but with the underlying representation visible. The database incorporates features such as long fields, long non-serializable (conversation) and nested transactions [DL88]. It is realized in a client-server architecture running over a local network. Database access is by interpreting explicitly represented type descriptor objects.

Our objective with this browser tool is to make the version space more easily accessible. While basic configuration control technology has improved to cope with demands for more flexible revision and variant support, facilities to allow the user to understand, manage and exploit this increasingly more complex versioned structure have not advanced significantly. As stated in [Rei89] “There is a very real danger that the data base of a project under IPSE control will become something of a ‘black hole’ as far as the user is concerned.” Information is difficult to locate, and when located, difficult to interpret. As experienced by many users, many engineering databases are mainly used as a passive box just logging the evolution. Users should rather be allowed to ac-
tively use the version space during exploration and cooperation.

In the following section the principles of change-oriented versioning and definition of multi-version contexts are outlined. Section Three describes product selection and navigation in the product space. When building visual representations, the browser uses information from both the version and the product space. Techniques available are presented in Section Four. Then the system architecture and the actual user interface of the browser is presented. Finally some remarks about support for cooperative work and the notification mechanism are mentioned.

2 Change-Oriented Versioning

Using the change-oriented versioning (COV) model [LDC+89], versioning can be considered largely orthogonal to data structuring (data model, schema, composition structure, etc.) as suggested in Figure 1. A version is global, i.e. a “view” of the entire database with any version-related variability removed. Each piece of information (a fragment\(^1\)) stored in the database is tagged by a visibility expression which denotes the set of versions in which the fragment will appear.

Each transaction must define a scope, an ambition, denoting the set of versions (not only one version) it wishes to influence or modify. When inserting a fragment, its visibility is assigned the transaction’s ambition, i.e. it will appear in all versions in the ambition. When deleting a fragment, its visibility becomes the set difference between the old visibility and the ambition, so that versions in the ambitions are removed from the visibility set. When modifying some value, the fragment is split into two. One fragment keeps the old value and its visibility is updated in the same way as when deleting. The other fragment with the new value gets a visibility equal the intersection between the old visibility and the ambition. Figure 2 shows how database operations are mapped to visibility updates and addition of new objects.

In addition to the ambition, a transaction can also define a subset of the ambition called the choice used when retrieving information from the database. As a special case, the choice can consist of a single version, and is then called complete. Complete choices are useful when checking out information to be used by external tools which are only able to handle a single version of some structure (e.g. a compiler).

![Figure 1: The product space and the version space](image1)

![Figure 3: The version space](image2)

Figure 3 illustrates the version space with the ambition and the choice of a transaction, and two example visibilities. Individual points in this pictorial representation correspond to complete choices. An object with visibility v1 will appear in the current version view, while objects with visibility v2 are not visible.

In the EPOS project we are using user-defined, global boolean variables called options to name versions and describe the variation of the product. An option denotes the inclusion or exclusion (if negated) of specific changes, and is typically related to the external properties of the product (a functional change). The ambition, choice and

---

\(^1\)In an E-R database the fragment size is typically an E-R attribute, except for long fields where a fragment can be a text line, a word or a character.
visibilities are boolean expressions over options\(^2\). New options are introduced as needed. The structure of the version space can be considered a \(n\)-dimensional boolean space with two values in each dimension, where \(n\) is the number of options defined so far. Multi-version ambitions and choices are expressed by \textbf{not} specifying values for some options, since this is treated as the the set of versions where these options are either \texttt{true} or \texttt{false}.

To simplify the use of COV for large systems, version description mechanisms [GKY91] have been designed. Abstract descriptions specifying desired properties and functional requirements are evaluated to an ambition and a choice. Suggested techniques include aggregates, constraints, validities (properties of versions) and preferences.

Figure 4 shows an example of versioning according to the COV model and a possible representation in a database. An entity 3001 with a single integer attribute is considered. Initially the entity is created and its attribute assigned the value 3 during a transaction with ambition \texttt{true} (a). An option \(a\) is introduced, and during its implementation 3001’s attribute is updated to 5 (b). Then a new feature is planned, and an option \(b\) is added. In the course of implementing \(b\), the attribute is updated to 7 (c). Finally, during a transaction to merge these changes (the ambition is \(a \land b\)), entity 3001 is deleted (d). In the figure the visibility expressions are simplified where possible, e.g. \(b \land \neg(a \land b) = \neg a \land b\).

For a database state like Figure 4 (d) and options \(a\) and \(b\), the possible versions and the corresponding value for 3001’s attribute are:

<table>
<thead>
<tr>
<th>OID</th>
<th>Visibility</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>3001</td>
<td>true 3</td>
</tr>
<tr>
<td>(b)</td>
<td>3001 (\neg a)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3001 (a)</td>
<td>5</td>
</tr>
<tr>
<td>(c)</td>
<td>3001 (\neg a \land \neg b)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3001 (a \land \neg b)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3001 (\neg a \land b)</td>
<td>7</td>
</tr>
</tbody>
</table>

\(^2\)Due to efficiency considerations, ambitions and choices are restricted to be conjunctions over options and negated options.
### 3 Product Selection

#### 3.1 Synchronization

A crucial feature of any DBMS is handling of concurrent multi-user access. The first observation is that versioning obviates the need for locking to some extent. Different transactions can access and updates different versions of objects without causing any conflicts. Secondly, when using the COV model, the ambition can be utilized for detection and possibly avoidance of update conflicts. Traditional "strict" concurrency control can be provided by enforcing non-overlapping ambitions since no change has any effect on versions outside the ambition.

Synchronization by means of ambitions can be augmented by conventional fine-grained locks if necessary. Still by only using ambitions, fairly precise control with possible update conflicts is achieved. Since ambitions are specified in advance, detection and possibly negotiation between transactions is possible. Also, conflicts are limited to versions in the intersection.

#### 3.2 Product ambition, watch-area and choice

In the product space, different areas can be defined to control and coordinate multi-user access. The product ambition is the maximum set of instances allowed to be updated during a transaction. It defines a sub-database to which the database could guarantee exclusive access if desired. Different approaches can be used to express and represent such a set of instances depending on the intended use.

In most engineering activities different kinds of hierarchies are regarded essential and much utilized. A system usually consists of multiple, intertwined hierarchies. These hierarchies are the main conceptual maps of the structure. Some examples:

<table>
<thead>
<tr>
<th>Choice</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬a ∧ ¬b</td>
<td>3</td>
</tr>
<tr>
<td>a ∧ ¬b</td>
<td>5</td>
</tr>
<tr>
<td>¬a ∧ b</td>
<td>7</td>
</tr>
<tr>
<td>a ∧ b</td>
<td>(entity 3001 does not exist)</td>
</tr>
</tbody>
</table>

- abstraction levels: aid comprehension and understanding.
- composition: describe the aggregation of large systems.
- dependencies: enforce consistency and support impact analysis and change propagation.
- classification: group items with similar properties, e.g. in object-oriented libraries or libraries of reusable components.

We have presently settled for a very simple mechanism to express regions in the product space. The user selects a root, and all entities computed by the transitive closure of a special composition relation from this root, and all relationships connecting these entities, are considered part of the product ambition.

The product watch-area is defined with an equivalent mechanism as the product ambition. Whenever database instances are created within this area, the transaction is notified.

A product choice is the actual set of instances currently in the view. This set is usually much smaller than the product ambition. In most cases the product choice is a subset of the product ambition, but not necessarily. The product choice is represented as an explicit set of instances. Navigation commands manipulate this set.

#### 3.3 Query mechanism

In an early prototype we implemented a query mechanism based on direct manipulation of a visual representation of the entity-relationship schema of the database (or a subschema thereof), similar to the approach in [CERE90]. Clicking on a symbol for an entity class, a relationship or a relationship role would toggle between dimmed and full intensity. The constructed query included all non-dimmed entity and relationship types. In addition, the query could be restricted by specifying for each type patterns for attributes to match.

Our experience indicated that this approach was an over-kill for the anticipated use in our problem domain. Although most users were quite familiar with the schema representation, they had difficulties in predicting the effects of their queries. Fre-
quently, users stated too wide queries and thereafter did a complete backtrack. Most queries were very simple, and surprisingly few different variants were actually used for a specific schema.

3.4 Navigation mechanism

The main use of the browser is to support the design, development and maintenance activities. A user typically browses around the database to localize a subsystem to start working on. Components in this subsystem are then edited, and the product rebuilt and tested etc. Investigation of different types of dependencies to predict consequences of changes must also be supported. Still another use is to examine all instances of a type to get some kind of global overview.

So rather than providing SQL-like query facilities, we mainly base our navigator on a step-by-step browsing mechanism. This is similar to point-to-point navigation supported by most object-oriented databases and hypertext systems, and is sufficient in most practical situations. A filtering mechanism allows the user to restrict which relationships to follow to a set of selected relationship types. Possible destinations can similarly be limited to certain selected entity types. This allows navigation by any hierarchy or any combination of hierarchies of the product.

The main navigation commands are:

- expand (follow a relationship)
  A relationship is followed from a source to a destination. The relationship and the destination is included in the product choice.

  The source can be a single entity, a set of entities, or all entities in the product choice. All relationships (of one of the currently selected relationship types) from the source are followed. Destination entities must be of one of the currently selected entity types. It is possible to expand one step, n steps, or the full transitive closure.

- remove
  An instance or a set of instances are removed from the product choice.

- search
  The set of instances matching the stated patterns are included. This mechanism can be used stand-alone, or in connection with an expansion to further limit the query.

- zoom into
  The interior of an object is examined. A window containing the attributes of the object is opened.

- backtrack
  Re-establish a previous product choice, thereby “undoing” one or several navigation operation.

In addition there are several convenience commands defined as combinations of the main commands. For example, the next command removes all instances not in the source set and expands one step.

4 Visualization

When multi-versioned structures are visualized, the visualization must convey information from both the version and the product space. Figure 5 shows the principle for how a visual representation is derived from an instance in the database. A version visualization process computes some visual attributes based on the objects visibility. Other visual features can be utilized to communicate properties from the product space, e.g. the type of the instance, values of its attributes, connections to other instances etc. There is a slight simplification here. In some aggregation techniques, both for product and version related information, data from several instances may be used to derive a single visual representation.

4.1 Product Visualization

Which visualization techniques are most suitable, will depend on the data model, the problem domain, intended task support etc. We have so far focussed on pure entity-relationship instances. Later we intend to include views for textual data and attribute name-value pairs. Figure 6 shows the primary techniques we have considered for E-R data structures.

In a graph view there is a vertex for each entity, and an edge for each relationship in the product choice. Affinity views [Pin90] differ from graph
views in the way relations are visualized. Relationships are not drawn, but instead an ideal distance between each pair of entities is computed as a weighted sum of values from the connecting relationships. The vertices are then given positions to reflect these distances in the best possible way. A **value** view is an aggregation technique where the instances in the product choice are partitioned according to some attribute value, and the group is given a single visual representation, usually with some visual feature to indicate the number of instances contained.

**Graph** views are most commonly used. **Affinity** views are convenient to obtain global overviews of complex domains (large number of relationships, big connectivity). **Value** views are a useful complement to the others by allowing grouping of entities by other criteria than the explicitly represented relationships.

### 4.1.1 Visual attributes

For each entity and relationship type, there is a mapping to a set of visual features used when making a visual representation, e.g., shape, color, brush, and fill pattern. Labels can typically contain the identifier or attribute values of the object. A powerful visualizer should also have mechanisms to allow definition of other visual attributes as functions of attributes of the database object.

A set of such mappings can be stored in the database, and are easily accessible in a menu. The user can create new sets or modify existing ones in an editor window.

#### 4.1.2 Layout algorithms

Automatic algorithms for layout of the structures are essential for the operation of the browser. The structures are derived (through a query) and continuously updated, and there are possibly multiple views of the same instance in different contexts.

Different layout procedures can be used, depending on the characteristics of the structure. For the graph view we have implemented several algorithms, namely barycentric [STT81], median [GNV88], and Eade’s splitting [EK86] algorithms for hierarchical graphs, and a Kamada spring embedder [KK89] algorithm with optional orthogonalization for general graphs. For the affinity view we have an operational algorithm based on distance analysis [DLPT82] that provide optimal or near-optimal solutions. However, this algorithm is too slow to be used interactively since matrix diagonalizations and eigenvalue computations are involved. Therefore we plan to switch
to an adapted spring embedder algorithm. The value view uses a very simple tree layout algorithm.

Since the product choice is incrementally changed as the result of navigation commands, layout procedures must operate incrementally to avoid loss of orientation. In this regard graph stability is important, i.e. to keep as much as possible of the graph layout unchanged despite structural changes.

4.1.3 Graph abstraction

For the graph view, the ability to temporarily collapse a set of vertices to a single vertex is useful in many situations. Edges incident to vertices in the set are then drawn as incident to the super vertex. [NPT90] defines three types of graph abstraction: black box, grey box and white box, depending on the super vertex contents. However, due to the complications of handling graph abstraction in combination with incremental layout, we have not incorporated it so far.

4.2 Version Visualization

We are currently designing and analyzing a number of techniques to visualize multi-versioned structures. Depending on the properties of the structure, the capabilities of the output device and the characteristics of the supported task, different approaches can be preferred. For each technique, different visual features can be used to mediate version-related information.

difference The degree of difference between two specific versions is visualized. Use: locate differences, locate the most different parts.

affected The degree of modification during a specific transaction is visualized. Use: locate errors introduced in a specific transaction.

entropy Visualize the degree of variance for each component for a set of versions. Use: identify “problem” (often changed) components.

age Visualize the time since the last modification of a component. An intuitive effect can be achieved by using colors: bright red for recently changed parts towards blue for more stable parts. Use: identify stable and evolving components, locate current activity.

probability Visualize the probability of occurrence for each component for a set of versions. Use: comprehension and general navigation.

merge Use an independent visual attributes for two specific versions. Fragments appearing in both or in any one of the versions are easily identified. Use: merge two versions, or migrate changes.

compatibility For each pair of components, compute the degree of overlap between their visibilities for a set of versions. Use statistical distance analysis [DLPT82] to map this \( n \times n \) matrix down to a single dimension controlling a visual attribute. Visually similar components are likely to have occurred in the same configurations. Use: compatibility of components.

structure variance Use a 2.5 dimensional view with the version space into the paper plane. The variance or evolution of a set of relationships over a linear sequence of versions is visualized. Use: detailed knowledge of the variance of relationships for a small number of versions.

backbone structure For a possibly large set of versions, visualize the probability of occurrence for each relationship. A lower cut-off value and a logarithmic scale is recommended. Use: extract the invariant part of some structure.

5 Architecture

Figure 7 shows the overall architecture of the database system and the graphical browser. Starting from the top of the figure is the versioned database which provides persistent storage of multiple versions of entities and relationships. There can be multiple transactions connected to the database, multiple product selections for each transaction, and multiple views for each product selection. As opposed to many other database systems supporting versioning, a COV-based database performs version selection prior to product selection.
Figure 7: System architecture
6 The User Interface

The design of the browser's user interface reflects the structure of the architecture. There is a transaction window for each open transaction, a product window within the transaction window for each product selection, and a view window for each view of a product selection. Nesting of windows is used to indicate the hierarchy.

The browser's user interface must provide facilities for easy manipulation of each of the three contexts defining its operation. In the transaction window there is an option menu to set the ambition and choice. The product ambition and product choice are updated according to the navigation commands issued in any of the views of the product selection. The visualization specification is registered in pop-up menus in each view window.

The browsing actually takes place in the drawing area of each view. In addition to the product navigation commands which can be issued here, a number of view-specific operations are available. These are operations that only affect the pictorial representation, without changing any underlying objects or contexts. Panning and graphical zooming are available via standard user interface components. Vertices can be manually moved by use of the mouse, and connected edges follow. Visual attributes of individual symbols can be changed. Selection allows easy application of operations to a range of instances.

In Figure 8 two transaction windows are shown, each with an option menu on the left. An option menu consists of two columns of three-state pushbuttons, the left one specifies the ambition and the right one the choice. Transaction 1 has ambition sun.os ∧ Motif, while transaction 2 has ambition sun.os ∧ bugfix138 ∧ ¬ mouse.sel. An unset or non-determined state of an option is indicated by a dotted appearance. In the transaction 2 window an expansion filter menu is also shown. Here the user can select (or de-select) which relationship types to follow and entity types for destinations for the expansion command. Transaction 1 has two product selections: one with the product ambition root /prosj.x/ui/ged and the other with root /prosj.x/dbif. The first product selection has also two views, a graph view and an affinity view. At the time of this snapshot, the user is about to change the visual attribute parameters for product visualization for the graph view.

7 Cooperative work

Parallel development is often necessary in design tasks due to the amount of work and strict time schedules. A versioned database can support this progress of development in a controlled way. Between different activities, various degrees of cooperation can be desired. In a COV-based database, we specify these policies by means of ambitions and choices, or for large projects by validities in addition.

Support for cooperative work is realized by means of overlap between ambitions, both in the version and product space. Ambition overlap is regarded as intended cooperation between multiple transactions. Note however that the exact behavior is governed by both the ambitions and the choices of the participating transactions, as indicated in Section 7.2.

7.1 Concurrent access

Since there is no support for conventional locking in the database currently, transactions are made aware of ambition overlaps by a notification mechanism. If exclusive access is essential to preserve data integrity, the ambitions must be reduced, or the transaction must negotiated with the other transactions to get them to reduce their ambitions.

7.2 Notification mechanism

Notifications are events sent from the database process to the connected transactions to coor-
Figure 8: User interface of the browser prototype
dinate and synchronize work between multiple transactions. Their generation is caused by operations issued in the database. A transaction can ignore or react to a notification at its own will.

There are different types of notification:

- transaction related: open/close transaction, change ambition or product ambition.
- version space: create a new option.
- product space: create/delete/update an entity or a relationship, create a new entity or relationship type.

Together with the notification type, an identifier of the issuing transaction and an identifier of the affected object (if any), are transmitted.

The propagation strategy (i.e. what transactions are notified) depends on the notification type. Transaction-related notifications are propagated to transactions with overlapping ambitions. Creation of new options and new types are broadcast to all open transactions. Creation of entity or relationship instances are propagated to transactions with choice and product watch-area overlapping the visibilities of the new instances. Deletion and updating of instances are propagated to transactions where the instance is inside the choice.

7.3 Browser behavior

The browser uses the notification mechanism to keep all views consistent with the database contents at any time. Changes done by other transactions are immediately propagated and made visible in the view if the affected object is in the current context.

For deletion or modification, the visible representation of the object is updated according to the changed state. New instances created are inserted in the product choice and shown on the screen, if the new instance is inside both the choice and the product watch-area. When new options or new types are created, the browser updates its internal tables used for the menus.

Note that deleting an instance does not necessarily imply that the corresponding visual object will be deleted. The object in question is deleted for some versions, but depending on the ambition of the transaction issuing the operation and the choice of the this one, the object might or might not still be visible. This illustrates how COV permits specification of different patterns of cooperation.

8 Conclusion

The browser is being implemented to run under the X Window System. It is implemented in C++ using the InterViews [LVC89] library to construct the users interface and the graphical interaction parts. The layout algorithms are implemented, but some of the view generation techniques and user dialogs still remain. The COV-based entity-relationship database is operating. The next implementation task will be to extend the techniques outlined here to also cover textual data (longfield) and name-value attribute pairs defined in our data model. Updating (editing) facilities are also needed.

During the design of this browser flexibility has been emphasized. A single technique cannot be equally appropriate for the diverse range of users and for different problem domains. A powerful browsing tool must provide a wide range of techniques and customization possibilities. Our browser allows users to compose their favorite views by a combination of visualization techniques and sets of preferred visual features, and store and retrieve these definitions symbolically.

Adaptation of the presented techniques to other models (data models, version models) are possible. However, the uniform treatment of versioning in COV and its orthogonality to the product structures, simplifies the realization since each information space can be treated separately.

9 Acknowledgments

We wish to thank Reidar Conradi, Even-André Karlsson and Anund Lie for interesting discussions and comments. The helpful comments of Dr. Richard Cooper and the anonymous referees are also gratefully acknowledged.
References


