Database Support for Evolving Design Objects

Alexandros Biliris[1]
Computer Science Department
Boston University, Boston, MA 02215
E-mail: biliris@bu-cs.bu.edu

ABSTRACT

One of the most important problems of database management systems for CAD applications is support for evolving design objects. In this paper we provide an integrated solution for three very important issues: design project decomposition into subprojects, design synchronization among engineers working on different parts of a large design, and documentation of the design process. We discuss the interaction of versioning with time and we show how these two orthogonal concepts can be used in the documentation of the design process in an engineering design environment.

1. Introduction

A CAD product is an aggregation of design objects and associated documents. For instance, a product in a mechanical application is associated with a variety of drawings such as detail and assembly drawings, exploded assembly drawings, etc. These design objects are frequently called *representations.* A design object may have many versions. Each version represents a particular description of the design object as it has been defined by a user at some point in time.

A version of an object may reference any number of objects that in turn may be referenced (shared) by any number of other objects. Such objects that are composed of other objects are called *complex*, [Lori83], or *composite*, [Bane87], and the hierarchical composition of a complex object is referred to as the *configuration hierarchy* of the object.

The design process starts when some requirements and specifications for a particular design object are defined to be met. Design libraries in a CAD installation contain objects that are used in the construction of new objects, which in turn may be included in that library. If some parts of the object under development have already been designed as part of a previous project, then they need to be integrated in the larger design. When none of the existing library components match the desired behavior for some of the components of the larger design, the construction of these components proceeds in *parallel* with the larger design.

[1] This research was supported in part by Computervision, a division of Prime Inc., under contract AA290424.

This design activity must somehow be coordinated. In this paper we describe an integrated mechanism for a database system to support the following.

Design project decomposition. Typically, teams of designers are needed to design a large product. The design project is decomposed into smaller subprojects until a fairly manageable and well specified piece of work can be defined. The design activity of each individual designer to whom such a piece of work is assigned may produce many versions. Since the design of components proceed in parallel by different designers in different subprojects, a version being designed in one subproject may reference objects that are themselves under construction in another subproject. This brings us to the next issue.

Design synchronization. Traditional database systems provide an inappropriate transaction model in that they support a large volume of short duration transactions (few seconds). Design applications need support for a small volume of long duration transactions lasting days. Applying the same transaction techniques to design objects, it would severely restrict the opportunity for parallel design,, [Lori83]. More specifically, users should be able to synchronize their designs by being aware of new (perhaps incomplete) versions rapidly as they are being produced by other designer, and at the same time prevent premature disclosure of a version, that is shared among users working on the same project, to other designers.

Design documentation. A person being involved in the design process for a particular product should be able to look back in time and see how a design was evolved. More specifically, users must be able to

- track the evolution of a design by 'walking through versions' of this design, and also
- track the evolution of a *single* version of a design object over time.

1.1. Related Research

A generally accepted system architecture, for performance reasons, of an engineering design system is the one where a central server is connected to powerful workstations through a local area network [Katz84, Kim84, Lori83]. In many proposals this model has been extended so that a hierarchy of (logical) databases can be established between the server and workstations, [Kim84, Banc85, Klah85, Chou86, Ditt87, Katz87, Bili89a]; these databases are called *group*, *project* or *semi-public* databases. Some of these models, augment the transaction manager with new mechanisms to handle design transactions. [Kim84] couples project databases with nested transactions, and [Klah85] introduces a new kind of transaction, called group transaction, to coordinate teamwork. In [Banc85] each project database corresponds to a set of cooperating transactions that consist of a set of short duration subtransactions. On the contrary, in [Lori83, Ditt87, Bili89a] complimentary mechanisms, on top of existing transaction mechanisms, are used for access control and synchronization.

An important issue that research in this area has identified is the need for two kinds of reference: *static* and *dynamic* (also called, *generic*), [Atwo85, Bato85, Chou86, Ditt88, Bane87, Beec88, Land86]. A *static* reference is a reference to a *specific* version of an object. A *dynamic* reference refers to a *generic* object that somehow represents the set of versions of the (semantically) same object; the exact version is left unspecified. Thus, a design object may reference an object x that is itself under development without prematurely binding this reference to a specific version of x that happens to be currently available. Instead, a *context* (function) is used to map x to one of its versions, [Chou86, Beec88, Ditt88]. The set of contexts currently active in a particular workspace defines the *environment* in which designs are materialized in this workspace. A user can establish a number of environments and, by switching from one environment to another, can see various alternative configurations of the design version.

1.2. Outline of our Work

The paper is organized as follows. Section 2 briefly describes some data modeling issues related to versioning; [Bili89b] provides detail description of the data model. In Section 3 we review our previous work on project decomposition and synchronization, [Bili89a]. We describe the design database hierarchy that represents the subprojects of a large project. This hierarchy in turn is used to define the rules for access control. Updates (through the checkout/checkin mechanism) are always performed on new versions, [Rehm88, Ditt87]. In effect, concurrency control problems caused by multiple writes are transformed into problems of version management rather than concurrent access. Versions are entered into the database and become accessible to other designers in a well controlled way.

Section 4 discusses documentation of the design process. Keeping old versions intact enables designers to go back and start working from a previously designed version (not necessarily the last one). Clearly, design versions can not been seen as a linear list of states of an object over time. Thus, all of the solutions we are aware of maintain a hierarchy to indicate the *derived-from* relationship among versions. However, it is not sufficient to know all versions of an object as they are now. We need also to know how these versions were at some point in the past. Thus, in this section we introduce time in an orthogonal dimension to versioning. We explain our choices and provide implementation mechanisms for temporal data associated with versions. Finally, in Section 5 we summarize our work.

2. Data Model

The database is a collection of (database) objects, each identified by a system generated id, called *oid*. Every object is of a given type; it is said to be an instance of that type. The type provides the attribute specifications shared by all of its instances. Each attribute specification defines the type of the attribute value. An object maintains its own storage for the values of the attributes defined in its type. The *state* of an object is given by the values of these attributes at any point in time. A type constructor is provided so that an attribute value can be defined to be a reference (simple or part-of) to an object of a given type.

A type T can be defined to be *versionable*; a user can define the attributes that a generic object of this type must have as well as the attributes of the versions of this generic object. In effect this creates two types: T, whose instances are generic objects, and TVersion whose instances are version objects. Generic object attributes include, in addition to the ones defined by the user, a default version, the number of versions, a version descriptor for each of its versions, etc. The attribute values of a generic object are shared by all of its versions. For example, if an AdderChip (a generic object) has four pins, then all its versions have four pins too. This kind of sharing between an instance of T and its versions in TVersion is enforced by the model. The reader is referred to [Bili89b] for a complete description of the model. For our purposes, this minimal definition is sufficient.

A version is identified (at the language level) by the name of its generic object suffixed with a version number. Version numbers are not reused and they are assigned to versions when they are created. x.V1 indicates version 1 of object x.

3. Decomposition, Synchronization

For each project in a particular installation there is a *design database* to store versions produced by designers working on that project. The design database hierarchy represents the work breakdown of a large project into smaller subprojects that in turn are subdivided into smaller tasks until a fairly manageable piece of work can be defined. Each such piece of work is then assigned to a design engineer. Thus, leaf databases in the hierarchy represent *private* databases while the non leaf represent *project* databases. Design databases form a directed acyclic graph (DAG), not necessarily a tree structure. In the following we shall use the term database to mean design database; the term *shared database* refers to the collection of project databases.

In addition to the role of the design database hierarchy to describe the work breakdown of a large product, it also encapsulates information about access control, i.e., read and write rights on versions and version ownership. Note that to handle these rights on disjoint objects is quite easy. But our objects participate in the configuration of many objects. Thus, we have overlapping (shared) objects and a clear understanding for access control is needed.

3.1. Access Control

Every designer is assigned (owns) a private database and there is an owner for every database. Each version has a single *home* database as its main residence; multiple copies of it may exist in other databases. As the version is being moved from database to database, its home database changes too. Version ownership is *exclusively* determined by its home property; the owner of a version's home database is also the owner of this version.

Designers in a cooperative design environment need to exchange their designs for partial evaluation, even though their designs are *incomplete*. Clearly, exchange of incomplete designs is meaningful only among designers working on the same project, i.e., designers who are direct or indirect members of a common database. This is the reason why the database hierarchy establishes the following visibility rule: a version V is *visible* (i.e., it can be referenced or read) from a particular database *d*, if V's home database is *d* itself or one of *d*'s direct or indirect parent databases.

Write access rights are granted on a generic object, not for each individual version. A private database *d* can checkout a version from an indirect database *p* only if all databases in the path from *p* to *d* grant this right to their immediate children in that path. Note that a database may grant checkout rights to more than one of its children databases at the same time. The purpose of granting these rights is not to somehow control concurrency. Updates are always performed on new versions of the original version.

3.2. Version Mutability

The mutability of a version is indicated by its *status* which can be *in-progress, stable, frozen,* or *released.* An *in-progress* version resides in a private database *only* and thus it is visible to its owner only. Any operation can be applied to an *in-progress* version, including deletion, modification, etc. When a version becomes *frozen* all dynamic references are mapped to specific references and the version can not be altered in any way. A *stable* version is like a frozen version except that its generic references are not statically mapped; the binding is done at access time. A stable or a frozen version can be deleted by its owner. Finally, a *released version* of an object can not be deleted or modified. The status of a version restricts also the status of its referenced versions. The following table summarizes the characteristics of versions with respect to their status.

	in-progress	stable	frozen	released
readable	yes	yes	yes	yes
deletable	yes	yes	yes	no
context sensitive	yes	yes	no	no
modifiable	yes	no	no	no
status of referenced versions	any kind	stable, frozen, released	frozen, released	released

3.3. Design Sessions

A designer starts interacting with the shared system by initiating a design session. He then may *checkout* a version *x*.Vn from a project database. This operation creates a new *in-progress* version *x*.Vm, a copy of *x*.Vn, in his private database. The new version is considered to be *derived from* *x*.Vn. This relationship among versions forms a hierarchy, called *version derivation hierarchy*. Many versions can be derived from the same version. If the checked out version was stable, its generic references are mapped to specific versions at checkout time and in the environment defined at the private database. Many subsequent checkout operations can be performed on another or the same object. When the engineer is satisfied with the results of his design, he places some of these versions (not necessarily the last one) back into a project database for further consideration by performing a corresponding *checkin* operation. A *checkin* of a version *x*.Vm to a project database *p* makes *p* the current home of *x*.Vm. The version can optionally be specified to be frozen as a parameter of the checkin operation (the default status is stable); alternatively, the current owner of the version can promote it to frozen (and subsequently to released) at any time. When the version becomes frozen each reference to an object of versionable type is mapped to one of its versions that are currently available. The current owner of a version can *move* the version to another database; the status is not affected by this operation. A designer can cancel the checkout of a version by issuing an *uncheckout* operation which removes from the database the fact that someone has checked out a version and removes the version itself from his private workspace.

A design session spans many days and logins/logouts. When the designer ends the session, an implicit uncheckout operation is issued for all objects that have been checked out in his workspace but not subsequently checked in. Note that there is no way to undo a checkin operation. The design session is not the unit of recovery, [Lori83, Eckl87, Rehm88]. If such an action is required the current owner of the checked in version should explicitly deleted it.

4. Versions and Time

A user being involved in the development of a design object can see how this object has been evolved by following successor or predecessor versions in the derivation hierarchy. This hierarchy in conjunction with the version creation time associated with each version provides a historical picture of when and from which version a particular version V was created. Although, this is a useful piece of information, it does not tell us perhaps the most important thing: how was that version V at some point in the past. Clearly, we are not interested to track every single change performed on a version in a private database; that would be impractical. However, this problem exists even for versions that have been checked into the shared system and therefore can not be modified. The problem originates from the fact that versions in the shared system might be stable and as such they contain generic references that might be resolved to different versions at different times. This is illustrated in the following example. Consider a stable version *x*.V referencing object *y* (a generic object with one version, *y*.V1), and that generic references to *y* are resolved to *y*'s most recent version. Assume that *y*.V2 becomes visible to *x*.V at time *t*. A reference today to *x*.V maps *y* to *y*.V2 whereas if this reference was evaluated at

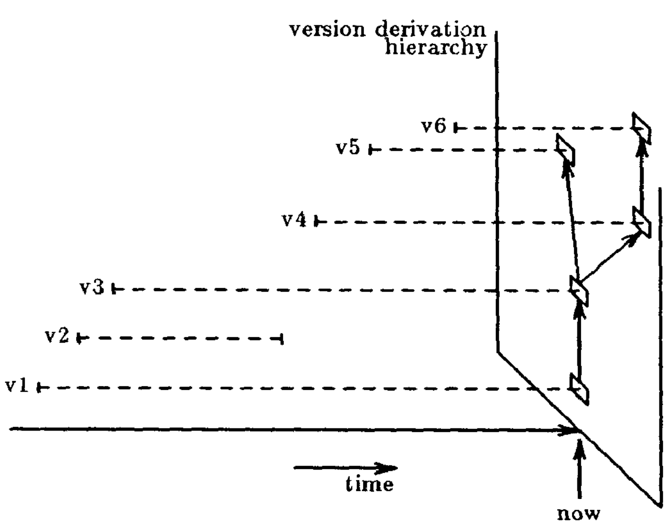

Figure 1. The evolution of a design object can be tracked either by walking through the derivation hierarchy, or for a single version by walking through time. V2 has been deleted at some time prior to 'now' and thus, it is not included in the current derivation hierarchy.

any time prior to t, it would have been mapped to the then most recent version of y, i.e., y.V1. Clearly, the time at which a reference is evaluated can alter the actual version of the referenced component. Therefore, version identification alone is not enough to fully describe a version object in the past; a temporal dimension should be introduced. Figure 1 shows the life span of each version of an object and its derivation hierarchy.

We are interested to track the evolution of a design in two dimensions; by following predecessor/successor versions in the derivation hierarchy, and by tracking the configuration hierarchy of versions in the shared system as of some time in the past. Our goal is to provide an understanding of how this can be economically achieved in CAD applications and not to present a general purpose high level language that deals with time (see [Snod87, Clif85, Gadi88, Caru88]).

4.1. Temporal Semantics

The subtle semantic difficulties caused by the introduction of time are discussed in this section. We provide a classification of attributes which is an augmentation of the time related concepts presented in [Snod87, Clif85, Sege87] with versioning.

- A *time attribute* is an attribute whose value type is time. The value of such attribute is a kind of time information that is defined by and it is of interest to the user only; it is called *user defined time* in [Cope84]. For example, 'creation date' and 'review date' are time attributes while 'designer name' is not a time attribute.
- A *temporal attribute* (TA) is an attribute whose value is a function of time. If the attribute value does not change over time it is called a *constant* attribute. For example, 'salary' is a temporal attribute. On the other hand, the version number of a version can not change over time and therefore this is not a temporal attribute.
- A *version attribute* (VA) is an attribute whose value type T is versionable. A VA's value can be a reference either to a generic object of T (a generic reference) or to a version object of TVersion (a static reference). In the first case the context mechanism is used to resolve the reference. Note that a type whose some of its attributes are VAs is not necessarily a versionable type.

The first two as well as the last two attribute types are orthogonal, and the set of version attributes and the set of time attributes are mutually exclusive, see Figure 2. For example, the 'home' attribute of a version is a temporal attribute, 'creation date' is only a time attribute, while 'date of last review' is both. Similarly, a version attribute may be either temporal or constant attribute. Note, that an attribute A of T1 of value type T2 can be defined to be temporal when T1 is defined. This is not true for version attributes; A is a VA iff T2 is versionable.

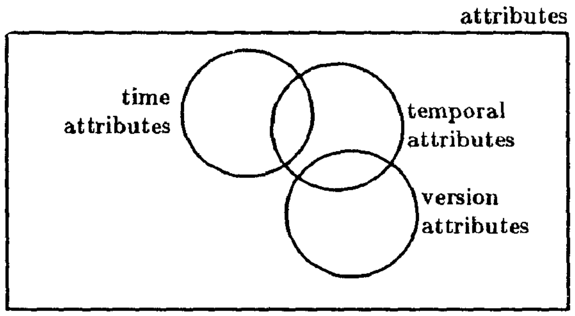

Figure 2. Classification of attributes

If time is going to be supported, the next question is what kind of time. Research in the area of time in databases has identified two important kinds of time [Snod87]. *Valid time* corresponds to the actual time at which an event occurs. *Transaction time* is the time at which information about a real world event is stored in the database, i.e., it is the time in which an event in the real world becomes known to the database. The example that is frequently given to explain the difference is the following. Assume the *salary* attribute of a person is given the value $1,000 on 1/80 (see the table below), and that this value is changed to $2,000 on 1/85. Now, suppose that on 1/86 the salary is to be changed to $2,500 but retroactive to 1/84. In other words, the salary value between 1/84 and 1/86 was mistakenly inserted in the database and this error was corrected on 1/86. Finally, the salary is increased to $3,000 on 1/87.

salary

valid-from	valid-to	tr-from	tr-to	value
1/80	1/85	1/80	1/85	1,000
1/85	1/86	1/85	1/86	2,000
1/84	1/87	1/86	1/87	2,500
1/87	–	1/87	–	3,000

The two kinds of time provide a distinction between what was *previously known* to be the state of the database at some time t and what is *now known* to be the state at the same time t.

4.2. Our Approach

First, we have chosen to deal with a system maintained time indicating when an update was registered in the database and therefore, we treat transaction time and valid time as identical. Our choice is justified for the following reasons. There is no difference between the time in which a real world event happens and the time that this event is known to the database. Updates (check-ins) on design versions can not be effective retroactively. The fact of the matter is that all design changes in a design version are performed to correct a mistake, something that was previously thought to be correct; however, it is exactly the versioning mechanism that deals with such updates (by creating new versions) and not some kind of temporal relationships. On the other hand, updates that become effective some time in the future, instead of the past, are both possible and desirable. It would be useful, for instance, to specify that a version will be frozen in two months from today. Such 'effective-time' attribute can be treated as time attribute that can be changed by the user at will.

Second, we have chosen to deal with *step-wise constant* data, [Sege87], where the value of an attribute at a given time *t*, is the one recorded in the database at a time closest to but not later than *t*. For instance, the 'count' attribute attached to an object, indicating the number of times referenced by other objects, exhibits the above behavior as illustrated in Figure 3.

Third, we want to have the choice to select the attributes of an object that must be treated as TAs. This is because there is no question that time support will be costly. It would be very unlikely that users will want all attributes of an object to be treated as TAs. Finally, we want to be able to group attributes that exhibit the same temporal behavior. For instance, when a review is completed, several attribute values are set at the same time, such as reviewer's name, review documents, suggested actions, etc. The semantics of such grouping is that the above modifications are timestamped together as a unit.

We require that access to current data to have the same performance as if time was not supported and this, regardless whether the accessed attribute is temporal or not. This is because queries accessing current data will substantially dominate temporal queries. Similarly, updates performed on non TAs must be as fast as if time was not supported. On the other hand, it is reasonable to expect that operations that involve searching or update on TAs will be slowed down.

4.3. Implementation Issues

An attribute (or a set of attributes) A is defined to be temporal at the type level. A set is allocated to hold temporal values of A; we call this set the *history vault* of A, HV(A). A single HV stores temporal data of an attribute for all objects of that type. Each entity in a HV(A) contains, in addition to the temporal values of A, the following fields:

oid	the system generated object identifier of the object whose A value is recorded in this entity.
time	the time in which the update is stored in the database
vflag	a valid-flag; a false value indicates that the value in the entity is no longer valid

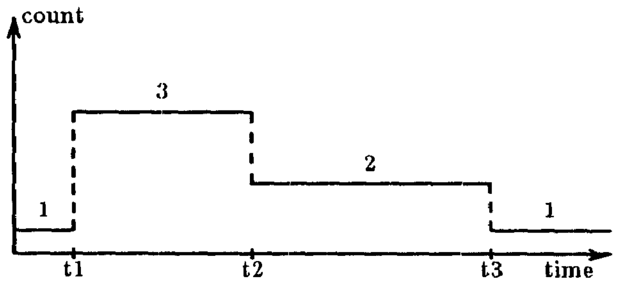

Figure 3. Step-wise constant data

Operations are performed as follows. When a temporal attribute A is updated, two operations take place. First, a new entity is inserted into HV(A) which contains the new replacement value and with *oid*, *time*, and *vflag* set to to the oid of the object, current time, and true, respectively. Second, the current value of A is overwritten to reflect the new value. When an object is (logically) deleted all HVs associated with the object's type are updated by adding a new entity with *time* set to current time and *vflag* set to false. The value of an attribute A of an object *o* as of some time *t* is found by searching the history vault of A for the entity whose *oid* is the oid of *o* and *time* is closest to but no later than *t*.

To view a version V as it was at some point in the past the user should set the system time dial to the desired time. If V was frozen or released at that time, no further searching in HVs is required. However, if V was stable, the system has to reconstruct the configuration hierarchy of V with the versions that were available and visible to V as of that time. To do this, the status and home database of each version, as well as the database hierarchy as of that time should be maintained in HVs. Note, that VAs need not be treated as TAs. VAs' values of a currently frozen version V are stored with the version itself. Since, the current version Vn of the referenced object is known, the system can find the generic object *x* of Vn and from that the version Vm to which *x* would have been mapped if V was accessed at that earlier time. Thus, VAs values as of some time can always be derived from the above information.

We emphasize that the storage strategy that will be used to store versions (e.g., see [Katz84]) is independent from the storage strategy of historical data related to a version. Accessing the current value of any attribute, temporal or not, involves no access to history vaults.

Finally, there is the indexing problem; we expect most queries against HVs to be multidimensional. Optical discs are excellent choice for archival storage; they are more reliable and less expensive than magnetic disks, they offer longer life expectancy than magnetic tapes, and they are easily removable to off-line and then back to the optical disk drive using some kind of mechanical robot (similar to juke-box). Although, the write-once characteristic of optical discs is not a problem for temporal data, it imposes some problems on indexing, [Ston87]. This is because, portions of the index may need to be moved to optical discs and most of the access methods require some kind of reorganization. We are planning to investigate multidimensional searching algorithms such as R-trees [Gutt84], and z-ordering [Oren86], in conjunction with Write-Once B-trees [East86], to see which one might be used easily on *both* medium.

5. Summary

In this paper we have given mechanisms to handle in an integrated way some of the most important problems of data management for CAD applications: synchronization and documentation. We discussed how the design database hierarchy can be utilized to provide a simple and meaningful way to designers to synchronize their designs, by controlling the visibility of each version they produce. Finally, we have shown how a temporal dimension should be introduced, in addition to the derivation hierarchy, to fully capture the evolution of a design object.

This project is part of an ongoing effort to analyze the requirements for data representation, constraint management and triggering, configuration management, and life cycle control for engineering data.

References

[Atwo85] Atwood, M. T., "An Object-Oriented DBMS for Design Support Applications," *IEEE, Proc. Computer Aided Technologies*, 1985, pp. 299-307.

[Banc85] Banchilhon, F., Kim, W. and Korth, H. F., "A Model of CAD Transactions," *Proc. Int. Conf. on Very Large Data Bases*, Stockholm, Sweden, 1985, pp. 25-31.

[Bane87] Banerjee, J., Chou H., Garza J. F., Kim W., Woelk D., Ballou N., and Kim H., "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Inf. Systems*, Vol. 5, No. 1, January 1987, pp. 3-26.

[Bato85] Batory, D. S., and Kim, W., "Modeling Concepts for VLSI CAD Objects," *ACM Trans. on Database Systems*, Vol. 10, No. 3, Sept. 1985, pp. 322-346.

[Beec88] Beech, D., and Mahbod B., "Generalized Version Control in an Object Oriented Database," *Proc. IEEE Data Engineering Conference*, February 1988, pp. 14-22.

[Bili89a] Biliris, A., and H. Zhao, "Design Versions in a Distributed CAD Environment," *IEEE, Int. Phoenix Conf. on Computers and Communications*, Phoenix, Arizona, March 1989.

[Bili89b] Biliris, A., "Management of Objects in Engineering Design Applications," BU, Comp. Sc., TR 89-005.

[Caru88] Caruso, M., and E. Sciore, "Context and MetaMessages in Object-Oriented Database Programming Language Design," *ACM SIGMOD Int. Conf. on Management of Data*, June 1988, pp. 56-65.

[Chou86] Chou, H., and Kim, W., "A Unifying Framework for Version Control in a CAD Environment," *Proc. Int. Conference on Very Large Data Bases*, Kyoto, August 1986, pp. 336-344.

[Clif85] Clifford, J., and A. Uz Tansel, "An Algebra for Historical Relational Databases: Two Views," *Proc. ACM SIGMOD Int. Conference on Management of Data*, Austin, Texas, May 1985, pp. 247-265.

[Cope84] Copeland, G. P., and Maier, D., "Making Smalltalk a Database System," *Proc. ACM SIGMOD Int. Conference on Management of Data*, June 1984.

[Ditt87] Dittrich, K., "Controlled Cooperation in Engineering Database Systems," *IEEE, Proc. Data Engineering Conference*, February 1987, pp. 510-515.

[Ditt88] Dittrich, K. R., and Lorie, R., "Version Support for Engineering Database Systems," *IEEE Trans. on Software Engineering*, Vol. 14, No. 4, April 1988, pp. 429-437.

[East86] Easton, M., "Key-Sequence Data Sets on Indelible Storage," *IBM Journal Res Dev.*, Vol. 30, No 3, May 1986, pp. 230-241.

[Eckl87] Ecklund, D. J., Ecklund E. F. Jr., Eifrig R. O., and Tonge F. M., "DVSS: A Distributed Version Storage Server for CAD Applications," *Proc. Int. Conf. on Very Large Databases*, England, 1987, pp. 443-454.

[Gadi88] Gadia, S. and Yeung, C., "A Generalized Model for a Relational Temporal Database," Proc. *ACM SIGMOD, Int. Conf. on Management of Data*, June 1988, pp. 251-259.

[Gutt84] Guttman R., "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Boston, June 1984, pp. 47-57.

[Katz84] Katz, R., H., and Lehman T. J., "Database Support for Versions and Alternatives of Large Design Files," *IEEE, Trans. on Software Engineering*, Vol. SE-10, No. 2, March 1984, pp. 191-200.

[Katz87] Katz, R. H., and Chang E., "Managing Changes in a Computer-Aided Database," *Proc. Int. Conf. on Very Large Databases*, Brighton, 1987, pp. 455-462.

[Kim84] Kim, W., Lorie, R., McNabb, D., and Plouffe, W., "Transaction Mechanism for Engineering Databases," Proc. *Int. Conf. on Very Large Data Bases*, Singapore, August 1984, pp. 355-362.

[Kim87] Kim, W., Banerjee J., Chou H., Garza J., and Woelk D., "Composite Object Support in an Object-Oriented Database System," *Proc. ACM, OOPSLA*, October 1987, pp. 118-125.

[Klah85] Klahold, P., G. Schlageter, R. Unland, W. Wilkes, "A Transaction Model Supporting Complex Applications in Integrated Information Systems," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Austin, Texas, 1985, pp. 388-401.

[Land86] Landis, S. G., "Design Evolution and History in an Object-Oriented CAD/CAM Database," *IEEE, Proc. COMPCON*, 1986, pp. 297-303.

[Lori83] Lorie, R., and Plouffe, W., "Complex Objects and Their Use in Design Transactions," *IEEE, Database Week - Engineering Design Applications*, 1983, pp.115-121.

[Oren86] Orenstein, J., "Spatial Query Processing in an Object-Oriented Database System," *Proc. ACM, SIGMOD, Int. Conf. on Management of Data*, Washington, D.C., 1986, pp. 326-339.

[Rehm88] Rehm, S., *et. al.*, "Support for Design Processes in a Structurally Object-Oriented Database System," *2nd Int. Workshop on Object-Oriented Database Systems*, Springer-Verlag, LNCS 334, 1988, pp. 80-96.

[Sege87] Segev, A., and Shoshani, A., "Logical Modeling of Temporal Data," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, May 1987, pp. 454-466.

[Snod87] Snodgrass, R., "The Temporal Query Language TQuel," *ACM, Trans. on Database Systems*, Vol. 12, No. 2, June 1987, pp. 247-298.

[Ston87] Stonebraker, M., "The Design of the Postgres Storage System," *Proc. Int. Conf. on Very Large Data Bases*, England, 1987, pp. 289-300.