OREGON STATE UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

Multiple Version Management of Hypothetical Databases

Earl F. Ecklund, Jr.

Darryn M. Price

Department of Computer Science
Oregon State University

Computer Research Laboratory
Applied Research Group
Tektronix Labs

84-40-1

Multiple Version Management of Hypothetical Databases

Technical Report No. 84-40-1

Earl F. Ecklund, Jr.

Darryn M. Price

Department of Computer Science
Oregon State University

Computer Research Laboratory
Applied Research Group
Tektronix Labs

July 16, 1984

Multiple Version Management of Hypothetical Databases

ABSTRACT

This paper presents a Hypothetical Storage Server for an experimental design database system. The storage server provides unified management of historical versions and hypothetical versions of objects in a design database. The extension of each database object is managed as a tree of multiple distinct representatives. One branch of the tree is designated as the primary branch, and its current representative is the primary version of the object. All other branches are considered hypothetical. A new branch in the tree is started when a new hypothetical version is derived from an existing representative. Hypothetical versions can be derived from any representative of the object, including prior versions of either the primary branch or a hypothetical branch. A branch grows when the current representative of the branch is updated. Both the primary version of the object and current versions of its hypothetical branches can be updated. Updating the primary version is equivalent to updating the object. An update to any other branch of the tree is a hypothetical update of the object.

Updates to the primary version of the object must be serializable, but derivation of hypothetical versions is not subject to such a constraint. Thus only write-write conflicts are subject to constraint, and conflicting updates can always be accepted by creating new hypothetical versions.

Multiple Version Management of Hypothetical Databases

Earl F. Ecklund, Jr.

Department of Computer Science
Oregon State University

Darryn M. Price

Computer Research Laboratory
Applied Research Group
Tektronix Labs

1. Introduction

In this paper we present HSS, a Hypothetical Storage Server, under development at Tektronix Inc., for an engineering design database system. HSS uses multiple logical representatives to store a database object and its hypothetical versions. The existence and availability of the multiple representatives is intertwined with controlling concurrent operations on these objects. Our approach contrasts significantly with the multiple version concurrency control or update algorithms that have appeared in the literature (e.g. [11]). In particular, we do not restrict concurrent transactions to be serializable[1] [3,5]. Two concurrent updates will produce two representatives for the object being updated. The representative created by the first update performed would be recognized as the current version of the object, while the representative created by the second update would be stored as a new hypothetical version of the object.

1.1 Background

An engineering design environment places requirements upon a supporting design database which are not met by traditional database management systems. Design actions

[1] Actually we do require that updates to the *current* version of an object be serializable, but we wish to allow parallel development of divergent (hypothetical) updates to the same object.

extend over a long period of time, perhaps days or weeks [7,8]. Multiple versions are needed to represent alternate designs or a history of released versions [6,10]. Multiple versions are also useful in supporting extended actions (i.e. long transactions) [2,4].

The relational model does not adapt well to the requirements of an engineering database [9,10]. Hypothetical versions can be represented using appropriately defined views [1,14,15]. In order to model the semantics of engineering design well, the relational model must be extended to capture more semantic information. Lorie and Plouffe [8] proposed representing complex objects with a *COMPONENT_OF(OBJECT)* domain to model hierarchical relationships. Stonebraker et al [13] propose to extend attribute types to model complex structure with abstract data types. Rehfuss et al [12] conclude that an object-oriented database provides the appropriate foundation for an engineering database.

1.2 HSS, an Object Manager

It is our thesis that an engineering database system should have two levels: an object management level and a semantic model level. Further we suggest that multiple versioning is best supported at the object management level. The semantic model must be cognizant of potential multiple versions, in particular of hypothetical versions, of an object, but multiple versions are nearly orthogonal to the semantics of structural complexity in an engineering database. Version management can reasonably be separated from structural and semantic modeling. Therefore, the Hypothetical Storage Server can and should be independent of the semantic data model.

HSS is being developed as a model-independent, object-oriented storage server providing unified management of hypothetical versions and historical versions of its objects. It provides a rich interface for a client data model to access or update the various versions of an object. Also, HSS exploits the availability of hypothetical versions to minim-

ize the ramifications of conflict among concurrent transactions. We allow any representative to be read and updated (albeit as a hypothetical update) at any time, so *a priori* concurrent read-read and read-write actions never conflict. Write-write conflicts can be avoided by creating a new hypothetical version with the result of the second update.

HSS should be useful to support client applications where hypothetical updates are to be investigated or where updates are performed interactively (e.g. by editing) over extended periods of time. HSS will provide a suitable environment for semantic data models supporting applications such as VLSI design, software development and documentation systems. In any design activity (e.g. VLSI design or software design) hypothetical versions have straight-forward application to alternative designs. Further, in software development, hypothetical versions should be used to develop revisions, leaving the current version unaffected until the revision is frozen and installed as the new current version of the object. Manual libraries, where manual pages are updated only when a command is enhanced, and legal case support systems, where hypothetical versions might be used for minority or divergent interpretations of a citation, are other examples.

2. The Structure of an HSS Database

A database stores information about a collection of entities. We define a database to be a triple, (*scheme, extension, mapping*). Intuitively, the scheme is a collection of entities that HSS manipulates at the request of its client data model. The extension is the set of representatives stored in the database for each entity in the scheme. The mapping identifies a unique representative for each entity in the scheme. We refer to the entries in the scheme as *names* and the mapping as the *name mapping* for the database.

We classify the entities as *objects* and *derivatives*. Objects are the principal means for access to and are the primary entities of the database. A derivative is a version

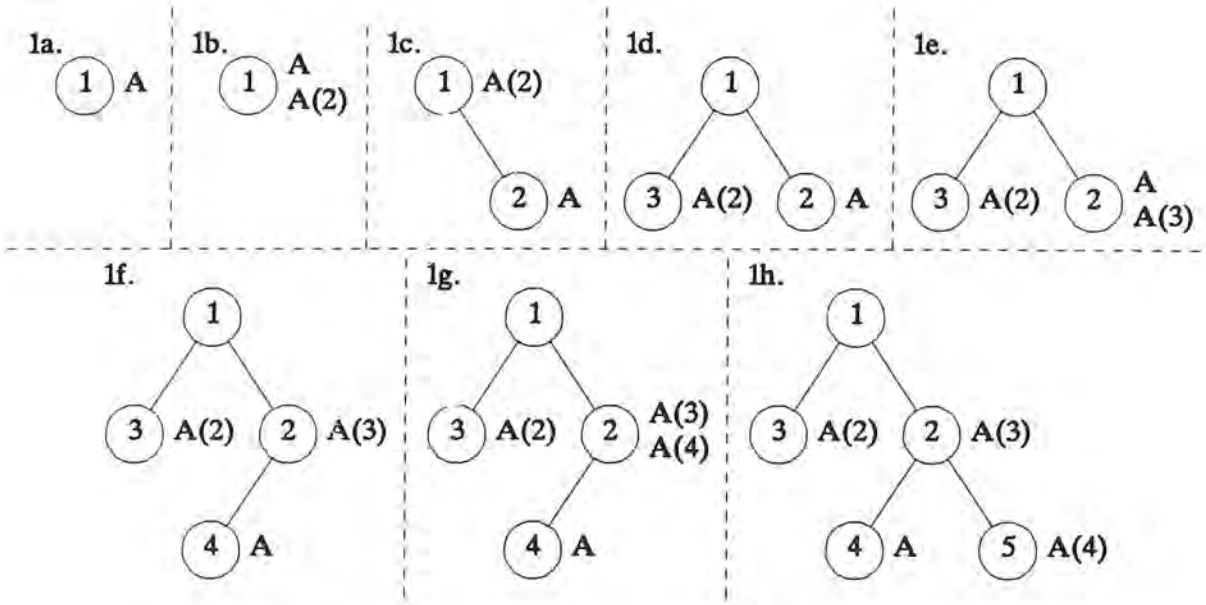
(actual or hypothetical) of its object, thus an object will be represented by one or more derivatives. Each object has one derivative, the primary derivative of the object, whose versions coincide with the object's versions. That is, the primary derivative of an object is the actual version of the object and each additional derivative is a hypothetical version of the object.

A *representative* is an instance of an entity. It is the basic unit of granularity for accessing the entity. That is, reads or updates are performed by atomically retrieving or writing the entire representative. A derivative is a series of representatives; successive representatives in a derivative are each obtained by update to its predecessor. An object can be regarded as a collection of multiple representatives, namely the union of the representatives of its derivatives. These are not redundant copies of a single instance, but a set of distinct, related instantiations.

2.1 The History Tree

An object is created with a single representative. Two operations, *update* and *derive*, are used to obtain additional representatives from the existing representatives for an object. Only *update* affects the current version of an object. *Derive* is used to initiate hypothetical updates, and several derivatives can be created from a single version of an object. Thus the history of an object will be linear with respect to the versions created by updates, but may be a tree when extended to include all derivatives and their representatives. The genealogy of the representatives of an object will be called the *history tree* of the object.

Figure 1 depicts an example of the evolution of a history tree. Note that in figure 1a A has one derivative, A(1), for which A serves as an alias. Note also that in figure 1g the derivative A(4) is derived from another derivative, and their representative is not the



The steps in the growth of the history tree for object A:

1a. Create object A from representative 1
1b. Derive A(2) from A, A(2) inherits representative 1
1c. Update A with representative 2
1d. Update A(2) with representative 3
1e. Derive A(3) from A, A(3) inherits representative 2
1f. Update A with representative 4
1g. Derive A(4) from A(3), A(4) inherits representative 2
1h. Update A(4) with representative 5

Figure 1. An Example of a History Tree.

---

current version of A. In fact, a derivative can be derived, at any time, from any representative of an object. For example, in figure 1h, it is possible to derive a new derivative, A(5), in such a way that A(5) will inherit representative 1.

A derivative is created by designating an existing representative as the source of the derivative. When creating a derivative, the derivative inherits the path from the root of the object's history tree to the representative designated in the *derive* command. Any update to the derivative will store a new representative that is exclusively a representative of the new derivative (at least until it is used as the source in a future *derive*).

In the history tree of an object, each branch gives the history of one derivative. That is, the path from the root to the leaf is composed of edges representing the *update* and branches representing the *derive* operations that produced the current version of the derivative from the initial version of the object.

A derivative may be created from any representative of its object. The new derivative will appear in the history tree as a branch (of length 0) started at the node for the representative from which it was created. Note that the current version of a derivative may be the source for a newer derivative. Thus while every leaf in the history tree will be the current version of a derivative, the current version of each derivative need not be a leaf (in the graph theoretic sense).

2.2 The Name Mapping

It is the fundamental responsibility of a database system to provide a well-defined mapping from its name space (i.e. scheme) into its extension space (i.e. representatives). That is, for each named object in the scheme, HSS must map consistently from the object's external name to a unique representative of the object. Our approach is embodied in the management of the names of objects, the representatives of those objects, and the mappings from the names to the representatives for the database system.

The name mapping is directly supported by the directories of the database system. Each external name requires entries that associate the name with its current version. In addition, each derivative will have an internal name and entries that associate the derivative with its current version. Thus an object with N derivatives will have N+1 directory entries, one for each derivative and one for the object. These entries will enable the system to directly access the current versions of all derivatives in the history tree of an object, and via the genealogy information of each representative, to access the entire his-

tory tree.

3. Some Semantic Issues

Our approach to object management is based on the paradigm of the database as a library. We view HSS as a repository for a collection of representatives. These representatives are to be checked out, used privately, and checked in (in good condition) to the library. (This is similar to the use of check out and check in by Lorie and Plouffe [8].) The HSS controls the flow of outgoing and incoming objects, monitoring their status.

Note that by returned in good condition we mean that the update is semantically consistent and valid in the context of the object being updated. Kutay and Eastman [7] conclude that "integrity does not exist for a database until design is almost complete." For this reason we have the client work on a copy of the representative being updated in his private file system until it has been restored to a state of semantic integrity. We do not assume that HSS is responsible for enforcing the semantic constraints of the application in which the database system is being used. Further, we assume that the client semantic data model will preserve the semantic integrity of the affected objects in the database.

When a client wishes to check out an entity, HSS provides him with a copy of that entity. The repository is presumed to have an adequate number of copies of each entity, so no client is ever kept waiting to check out an entity. When the client has finished using the entity he returns it to the library. If the copy checked out has been altered, the revised representative will be checked in to be used as the new current version of the entity. Similarly, if the client produces an alternate representative from the copy checked out, it will be checked in as a hypothetical version of the entity.

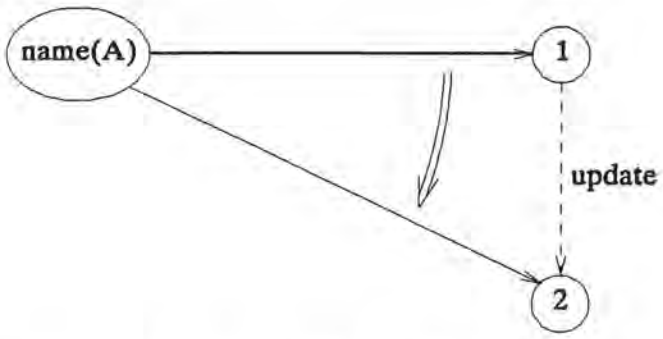


Figure 2. The Name-Mapping Transformation.

3.1 The Semantics of Updating

The fundamental role of a database system is to provide the mapping from the names of objects to their instantiations. An update (or derive) transaction determines a natural transformation of the existing name mapping. The transformation is made by merging the changes relating to the entities being updated into the existing mapping. Figure 2 represents the transformation determined by an update transaction.

The semantics of updating require that this transformation must be *effective* and *well-defined*. An update transaction to an entity is said to be effective if it causes a new representative to be stored in the extension of the entity. An update to an entity is said to be well-defined if the update is based on the representative that was the current version of the entity before the update. A transaction that modifies an entity must either be rejected or produce a new version of the entity.

The semantics of update must be adhered to. An update to an object is required to produce a new version of the object; an update to a derivative must produce a new version of the derivative. That is, an update to the principal derivative must produce a new current version, and should not produce any other representative nor have any affect on a non-principal derivative.

In a database system that provides serializability of update transactions, serializability implies each update is well-defined. In a multiple representative history tree, for a representative to be well-defined, the history of updates along the branch for one derivative must be a sequence of effective updates to the successive versions of that derivative. We wish to particularly emphasize that a new version of the object may *not* be obtained by performing an update to a derivative that is not the principal derivative of the object. An update to a non-principal derivative may not be retrieved under the name-mapping from the object name. An update to a non-principal derivative can only be retrieved under the name-mapping from its derivative name. (Using *assign*, the result of such an update may be made the new version of the object, but this must be done explicitly.)

3.2 Group Semantics

We anticipate that client data models will use multiple HSS objects to model a complex structure. For example, hierarchical designs might be modeled with one object which describes the structure and relationships among the components, and an additional object (or set of objects) to model each of the components. In this case, HSS must help the client semantic model maintain version consistency [4] among the set of objects used to model the complex structure. The group update operation will not update one current version in the group if it cannot perform all of the updates in an effective and well-defined manner as current versions of the same class of entities. In other words, group updates must be performed at the lowest common level of currentness. (See section 6.10.)

4. Entity Names

An object's name is used in two ways. Its extension is its entire history tree, and its instantiation is a representative that is designated as its current version. Designating an object's current version is done by associating the object with one of its derivatives and

instantiating the object with the current version of that derivative. Initially, an object X has one unnamed derivative X(1) which is the alias for the object. So long as no additional derivatives are created, the history tree for an object will be a serial history. This branch is the initial principal branch of the history tree.

The series of representatives along a branch of the history tree determines a sequence of time intervals corresponding to when each representative was the current version. We associate with each representative the time when the representative was produced. A version of an object that preceded the current version is referenced as X[t], where *t* is the time when the desired version was the current version.

A derivative need not be named, as it is accessible through the history tree of its object. For convenience, a derivative can be given an external name which will be part of the scheme for the object. All the derivatives of an object have implicit aliases denoted by subscripting the object name. Thus if the object X has N derivatives, they are recognized as X(1), X(2), ..., X(N), and any representative can be designated as X(i)[t] for appropriately chosen values of *i* and *t*.

An external name must be used to refer to a representative. External names may be of the following forms:

Entity_Name	refers to the representative that is the current version of the entity.
Entity_Name[time]	refers to a non-current version that was the current version of the entity during the specified time.

A valid Entity_Name may be of one of the following forms:

Object_Name	refers to the representative that is the current version of the object.

Derivative_Name	refers to the representative that is the current version of the derivative.
Object_Name(i)	where i is an implicit alias, refers to the current version of the i-th derivative derived for the object specified.

5. Directories

The third component of a database, the name mapping from the objects of the scheme to the representatives in the database, uses the directories of the system to maintain the information required to make the mapping. Three types of information are used: information about the entities (e.g. name, type of entity), information about the representatives (e.g. parent, address, genealogy), and information about the mapping (e.g. the current representative for each entity).

HSS uses two directories to keep the necessary information. The names directory contains information about entities: name, current representative and genealogy information. The locations directory enables representatives to be located, and root-ward tracing of the branch for a derivative. We will frequently refer to some of the attributes as being of type *token*. By a token we mean an internal name whose value is unique within the database system, and can be used as a key to identify each entity or representative.

5.1 The Names Directory

The attributes of an entity in the scheme are recorded in the names directory. The structure for the names directory is:

names(instance, name, object, derivative, representative, implicit alias, source)

Instance, a token, is the key used internally to identify the entity. Note that the instance token is immutable, and will be constant even if the name of the entity is changed. *Name* refers to the external name for this entity. Objects must have external names, but

derivatives may be unnamed (name = null). This attribute is the domain for the name mapping. *Object* is a token that identifies the object to which this entity belongs. *Derivative* is a token designating the derivative that supports the current version of this entity. Note that either the object token or the derivative token will equal the instance token, indicating that the entity is an object or a derivative, respectively. *Representative* is a token that identifies the representative for the current version of this entity. The *implicit alias* is used to distinguish derivatives. It is a number corresponding to the order in which the derivative was created. For an object, the implicit alias is interpreted as the number of derivatives created from the object. The *source* contains information representing the genealogy of the entity. In particular, it identifies the immediate parent of the initial representative of the entity.

5.2 The Locations Directory

For each entity, the name mapping must locate the representative of the current version of that entity. The token for the current version is in the representative field of the entry in the names directory. To complete the name mapping, the locations directory provides the address of each representative. The structure for the locations directory is:

locations(representative, parent, derivative, time, address)

Representative is a token that uniquely identifies each representative. This token is assigned when the representative is created, and is permanently associated with it. *Parent* is a token designating the representative from which this one was obtained (via an update). *Derivative* is a token that specifies the derivative to which this representative belongs. *Time* is the time at which the operation that created this representative took place. The time is used for path addressing of prior versions of an entity that were current at a specified time. *Address* is an implementation specific value indicating how to

access the representative.

6. Operations

The multiple representative structure for HSS requires several operations that manipulate the objects in the database. The required operations are: *assign, checkout, create, delete, derive, erase, name, read, return,* and *update*. In each of the following sections we discuss one of the operations and its role in HSS's multiple representative structure.

6.1 Assign

Each object's external name is mapped to a unique representative which is the current version of the object. Normal transitions of the name mapping are implicit in the semantics of the update operation. To make an abnormal transition, i.e., to designate a hypothetical version to be the current version of the object, the *assign* operation is invoked. The *assign* operation designates a derivative to be the new principal derivative of the object.

6.2 Checkout

To obtain a private copy of an entity, for update processing or extended browsing, the *checkout* operation is invoked. Since the copy may be returned as an update, a record of its source representative must be maintained. The *checkout* operation accomplishes this by prepending to the copy a header composed of the appropriate genealogy information. This genealogy information must not be altered while it is stored privately.

6.3 Create

This operation is used to create a new object within the current scheme. The initial representative of the object is obtained and its genealogy is recorded. A derivative is also created as the principal branch of the new object. Thus creating an object requires two

new entries in the names directory.

6.4 Delete

The delete operation is the inverse of create. It is used to delete an object from a scheme. This is done by removing the entries for that object from the directories of the database.

6.5 Derive

The *derive* operation is invoked to create a new derivative entity whose source representative becomes its current version. The new derivative is an entity with its own names directory entry that essentially points to the source representative. In order to update a representative, it must be the current version of some derivative. To update a representative that is not the current version of any derivative, or to update a representative of a derivative without altering its status as current version, one can derive a new derivative from it. An update to the representative via the new derivative will have no affect on the representative or older derivatives.

6.6 Erase

The *erase* operation is used to delete the current version of a derivative or to delete an *entire* derivative. In deleting any representative of an object, the system must guarantee that if another derivative has the designated representative in its branch, then the path through the representative will be maintained.

6.7 Name

A derivative can be identified in two ways, with its own external name or through its implicit alias within the object to which it belongs. The *create* and *derive* operations allow an external name to be specified at the time the derivative is created. If that

option is not exercised, then the derivative is unnamed and may only be referenced by its implicit alias. The *name* operation assigns an external name to an unnamed derivative or renames an object or derivative.

6.8 Read

To obtain a copy of an entity for browsing, the *read* operation is invoked. It is important to note the distinction between *checkout* and *read*. While *checkout* assumes that the user might update his copy of the entity and return it to the database, read does not. Thus the overhead of keeping track of the copy, and maintaining its accompanying genealogy information is unnecessary with *read*.

6.9 Return & Update

The *return* and *update* operations provide the "check in" function. The *return* operation indicates that a copy is now inactive and will not be "checked in" as an update.

The *update* operation is the backbone of the entire object management approach. *Update* is the only operation that extends a branch of the history tree of an object. It is here that the integrity of the update semantics is maintained.

The genealogy information in the header of the copy informs the system which source representative is being updated. The names directory is checked to validate that the source representative is still the current representative of the entity being updated. If so, the update is valid and processing continues. The new representative is stored and an entry in the locations directory is made. This new entry will have a unique representative token generated to identify it.

The names directory entry for the entity will be updated by setting the representative token to the token of the new representative. If the entity being updated is an

object or the object's principal derivative, the names entries for both the object and the principal derivative must be updated.

If the update was not valid, then an unnamed derivative is created from the source representative, and the update is applied to the new derivative. This is not an effective update, however, and a response is given that indicates the update failed and a new hypothetical version was created.

6.10 Group Updating

In the discussion above, we have directed our attention to operations as they applied to an isolated entity. In order to preserve semantic integrity, it is often necessary to update sets of related entities atomically. That is, either all entities in the set will be updated, or none of the updates will be applied to current versions of the entities.

In view of the semantics of updating, set atomicity is most important when the intent is to update each current version of a set of objects. In this case, either all or none of the updates must produce new current versions of their objects. If the intent is to perform a set of hypothetical updates then all of the updates must create a new version of a derivative which is not the principal branch of its object. That is, a set of hypothetical updates must affect only the hypothetical versions of the objects.

Therefore it is of paramount importance to determine if the sources for each object affected by a set update are still current. The update must be performed at the lowest common level of currentness represented among the sources for the set of entities involved in the update. If all members in a set of entities were current versions of objects when checked out and some are not current versions when the update is returned, the group update is not effective and an appropriate response must be given. There are four possible cases that should be dealt with as follows:

a) All sources are the current versions of their objects. Apply all updates to the current versions, producing a new current version of each object in the set.

b) All sources are still the current versions of non-principal derivatives of their objects. That is, each source is the current version of a derivative, but not the current version of its object. Process each update to produce a new hypothetical update of its derivative.

c) Some sources are the current versions of non-principal derivatives and some sources are prior versions of any derivatives. In order to update the prior (non-current) versions, new derivatives must be created from these sources. After the new derivatives have been created, then process the update as in case (b).

d) Some, but not all, sources are the current versions of their objects. In this case, all updates must by applied to non-principal derivatives. Thus new derivatives must be created from those sources that are the current versions of their objects. Then using these non-principal derivatives, process the update according to case (b) or (c) above.

7. Conclusions

We have outlined an approach to object management that would enable hypothetical updating of the objects in a database. The approach principally addresses incorporating a multiple representative structure into HSS, a hypothetical storage server. We also propose naming conventions that would affect the data model level. The system has several novel features principally related to the multiple representative structure. In this section we review some of the key observations presented.

HSS is an object manager that deals with multiple representatives for objects in a database. It specifically does not propose to support any particular data model. The most important of its features is the support for hypothetical updates, and the

incorporation of certain hypothetically developed entities into the current state of the scheme.

A multiple representation structure is developed for managing parallel development of both the current version of an object, and zero or more hypothetical versions of the object. The approach provides an optimistic checkout and update vehicle, in which no update is ever thrown away. A invalid update is applied to a derived representative in the multiple representative tree of the object.

The motivation for much of the design is derived from a "library paradigm". Careful analysis of the semantics of updating single entities, or groups of entities subject to a consistency constraint, yields insight into how the operations must behave. Specifically group update must execute at the lowest common denominator of currentness for the group of representatives being updated.

The interaction between multiple representatives and concurrency is richer than we first anticipated. In addition to implicitly affecting the concurrency control for HSS, the multiple representative structure may contribute to a more robust system. Typically, the recovery process reconstructs a representative from its parent. Does the presence of hypothetical versions provide a feasible alternate recovery path? This is a question that merits further consideration.

References

1. R. Agrawal and D. J. DeWitt, "Updating Hypothetical Databases," *Inf. Proc. Letters*, vol. 16, no. 3, pp. 145-146, 1983.

2. Duzan Badal, "Long-lived Trans. - Are They a Problem or Not?," in *Proc. Compcon Spring 1983*, pp. 503-507, IEEE Computer Society, 1983.

3. Philip A. Bernstein, David W. Shipman, and Wing S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. on Software Engineering*, vol. SE-5, no. 3, pp. 203-216, May 1979.

4. C.N.G. Dampney, "Precedency Control and Other Semantic Integrity Issues in a Workbench Database," in *Database Week Conf. on Engineering Design Applications*, pp. 97-104, IEEE Computer Society, 1983.

5. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Comm. ACM*, vol. 19, no. 11, pp. 624-633, November 1976.

6. Randy H. Katz and Tobin J. Lehman, "Database support for versions and alternatives of large design files," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 2, pp. 191-200, March 1984.

7. Ali R. Kutay and Charles M. Eastman, "Transaction Management in Engineering Databases," in *Database Week Conf. on Engineering Design Applications*, pp. 73-80, IEEE Computer Society, 1983.

8. Raymond Lorie and Wilfred Plouffe, "Complex Objects and Their Use in Design Trans.," in *Database Week Conf. on Engineering Design Applications*, pp. 115-121, IEEE Computer Society, 1983.

9. David Maier and Darryn Price, "Position Paper, Data Model Requirements for Engineering Applications," Tech. Rpt. CR-84-17, Tektronix, 1984.

10. Dennis McLeod, K. Narayanaswamy, and K.V. Bapa Rao, "An Approach to Information Management for CAD/VLSI Applications," in *Database Week Conf. on Engineering Design Applications*, pp. 39-50, IEEE Computer Society, 1983.

11. David P. Reed, "Implementing Atomic Actions on Decentralized Data," *ACM Trans. on Computer Systems*, vol. 1, no. 1, pp. 3-23, 1983.

12. S. Rehfuss, M. Freiling, and J. Alexander, "Particularity in Engineering Data," Tech. Rpt. CR-84-20, Tektronix, 1984.

13. Michael Stonebraker, Brad Rubenstein, and Antonin Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Data Bases," in *Database Week Conf. on Engineering Design Applications*, pp. 107-113, IEEE Computer Society, 1983.

14. Micheal R. Stonebraker and K. Keller, "Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System," in *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 58-66, ACM, 1980.

15. Micheal R. Stonebraker, "Hypothetical Data Bases as Views," in *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 224-229, ACM, 1981.