A UNIFIED FRAMEWORK FOR VERSION MODELING USING PRODUCTION RULES IN A DATABASE SYSTEM

by

Lay-Peng Ong and Jeffrey K. S. Goh

Memorandum No. UCB/ERL M90/33

27 April 1990

A UNIFIED FRAMEWORK FOR VERSION MODELING USING PRODUCTION RULES IN A DATABASE SYSTEM

by

Lay-Peng Ong and Jeffrey K. S. Goh

Memorandum No. UCB/ERL M90/33

27 April 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A UNIFIED FRAMEWORK FOR VERSION MODELING USING PRODUCTION RULES IN A DATABASE SYSTEM

by

Lay-Peng Ong and Jeffrey K. S. Goh

Memorandum No. UCB/ERL M90/33

27 April 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Unified Framework for Version Modeling using Production Rules in a Database System.†

Lay-Peng Ong & Jeffrey K.S. Goh

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, California 94720
U.S.A.

ABSTRACT

Most existing version servers are based on a common set of mechanisms and differ only in update semantics and terminology. Rather than introduce yet another version model, this paper shows how production rules in a database system can be used as the implementation mechanism for constructing version servers while retaining comparable performance to conventional version systems. This paper details the implementation of common version mechanisms using rules. By making the version server a high-level system, users have a logical and intuitive framework to implement their favorite version semantics. Thus, using rules rather than special purpose code is shown to be general enough to subsume existing version systems and is flexible enough to accommodate future extensions or new semantics.

1.0 INTRODUCTION

As pointed out in [KATZ88], despite the large number of proposals for version management systems, most of them are really only minor variations on a small number of themes. The main drawback of version systems is that the semantics have traditionally been hard-coded and thus these systems are difficult to extend when new semantics are desired. Furthermore, no single system has been general enough to subsume all of the proposals. Hence there is no single all-encompassing implementation. The purpose of this paper is not to introduce another model for version systems, but to show that production rules in a database system provide a framework for version modeling that can be used to implement the needs of any versioning environment.

Although this rule-based framework is presented in the context of the Postgres [STON 86] database system, the method is applicable to any database system that has production rules. As summarized in [KATZ 88], the major version concepts are:

(1) Versions using forward deltas
(2) Versions using backward deltas
(3) Version histories and graphs
(4) Equivalences

† This research was sponsored by the Army Research Organization Grant DAAL03-87-0083 and by the Defense Advanced Research Projects Agency through NASA Grant NAG2-530.

(5) Constraint propagation
(6) Change notification
(7) Change propagation
(8) Private, Group and Archive workspaces
(9) Dynamic and static configurations

In this paper, we will detail the implementation of each of these concepts using rules.

2.0 REQUIREMENTS

This section describes in greater detail the types of support an underlying database system should have in order to efficiently support the framework outlined in this paper. Except for the rules subsystem, absence of the following features does not preclude implementing the version system as described in this paper; however, having these features ensures that the implementation is at least reasonably efficient.

2.1 Historical Data.

By historical data, we mean those tuples which have been logically replaced/deleted over time and are not part of the "current" state of the database, but are instead part of some "past" database state. To efficiently support versioning, the underlying database system must be able to support queries on historical data. Simulating historical data using a conventional relational system is far too expensive. In Postgres [STON86], tuples are never physically overwritten or deleted. Instead, both the old and new versions of the tuples are retained, and the old tuple invalidated. Thus in Postgres, it is possible to query historical data. For example, the following time-range query finds all employees that earned more than $10000 in the 12th of January 1989:

```
retrieve (emp["Jan 12 1989"].all)
where emp.salary > 10000
```

This query is similar to a regular Postquel retrieve query except that the relation is qualified by a timestamp.

2.2 Rules.

Since the framework outlined in this paper is based on production rules, the underlying database system must be able to support such rules. The Postgres rules system (PRS2) [STON90] follows the production rule paradigm of a rule being an event-action pair where a typical rule looks like:

```
ON <event/condition> THEN DO [INSTEAD] <action>
```

If the "INSTEAD" keyword is used, the original query which triggered the rule is not executed, and the action of the rule is executed in its place. If "INSTEAD" is not used, then both the query that triggers the rule and the action of the rule are executed. In PRS2, an event can be any one of the usual retrieve, replace, append, or delete queries on a relation, and an action is any set of Postquel queries which optionally references the triggering event-relation. Postgres is not the only data manager that has a rules subsystem. For instance, Iris and Starburst both have similar rules subsystems. However, their triggering events do not include the "retrieve" event, and as such, they cannot support the versioning system we have described. We note, however, that adding "retrieve" to the set of triggering events complicates the rule evaluation model somewhat.

The following is an example of a rule in the PRS2 which moves an employee from the relation "emp" to the relation "old_emps" when he becomes 50 years old.

```
define rule same_salary
on replace to emp.age then do instead {

append old_emps ( name = NEW.name, age = NEW.age )
        where CURRENT.age < 50
        and NEW.age >= 50

      delete emp where emp.OID = CURRENT.OID
}
```

The semantics of PRS2 is that at the time an individual tuple is accessed, updated, inserted or deleted, there is a CURRENT tuple (for retrieves, replaces and deletes) and a NEW tuple (for replaces and appends). CURRENT is used as a tuple variable that refers explicitly to the current tuple in the relation that triggers the rule, and NEW refers to the new value(s) being appended/replaced.

2.3 Procedures

Hierarchical data structures can be supported in various ways. In Jhingran [JHIN 89], such support is partitioned into Value-Based, Object-Identifier-Based and Procedure-Based representations. Postgres supports the procedural representation where fields in a relation can contain procedural objects. These fields are called procedural fields. Procedural objects are executable programming constructs. In Postgres, procedures are collections of Postquel queries which can access the underlying database. Procedures lends a high degree of flexibility to the database schema, and allows hierarchical representation of data [STON 87]. Thus, in Postgres, a configuration is simply a relation where each sub-object is stored as a procedural field. Hence, when each sub-object is accessed, the respective procedure will be evaluated to produce the sub-object. We note that without similar support, the hierarchy needs to be flattened, making for an unnatural and space-inefficient representation.

2.4 Union Queries.

As will be shown in section 3.1, queries on versions implemented using forward deltas will be transformed by the into union queries. This section describes the semantics and optimization of union queries as it pertains to version processing.

First, let us denote the relational union operator (i.e., the union is on a per relation basis) by the vertical bar '|'. If we have a query of the form:

```
replace (A) where (A|B).name = "foo"
  and B.OID not_in C.DOID
  and (A|B).OID = A.OID
```

then the query will be split into the following two equivalent queries:

```
replace (A) where A.name = "foo"
  and B.OID not_in C.DOID
  and A.OID = A.OID                    (I)
and
replace (A) where B.name = "foo"
  and B.OID not_in C.DOID
  and B.OID = A.OID                    (II)
```

Using the rules of semantic optimization stated in [STON90], the semantic optimizer will remove all redundant clauses (clauses that have no link back to the target relation) and optimize query (I) to:

```
replace (A) where (A).name = "foo"
```

while query(II) can be dropped altogether. This is possible because of the special property of OIDs which ensures that (B.OID = A.OID) = NULL where B != A. Thus, the last qualification of query(II) will never be satisfied.

3.0 VERSIONS

While [KATZ88] describes a version as a semantically meaningful snapshot of an object in time, others have extended versions to include those that are not snapshots; that is, such versions should be optionally able to "see" changes in the base relation. The user decides at the time he creates a version whether or not the version is a snapshot of the base relation. In general, versions are maintained via differential files (deltas) [SEVE 76]. There are two kinds of deltas: forward and backward. Forward deltas are more suitable for implementing those versions which are "alternates" since they can optionally allow changes to be propagated to the "base" relation. Backward deltas, on the other hand, allow for cheaper access to the latest versions and are thus well suited for implementing versions which are "derivatives" of other versions. In this section, we show that both these methods can be implemented using rules.

3.1 Forward Deltas

Versions implemented as forward deltas are logical entities made up of 3 relations.

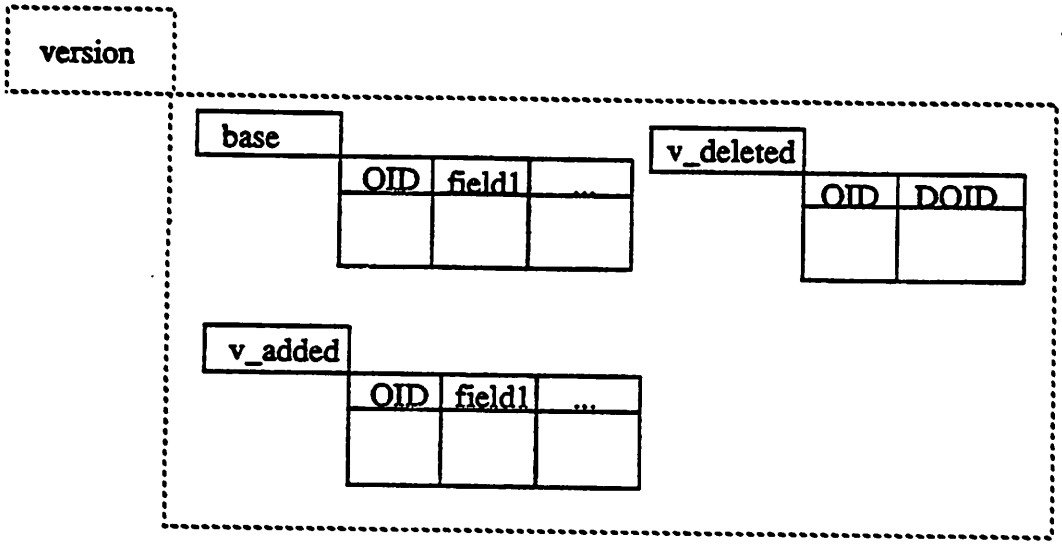


Figure 1 : A version is made up of three physical relations.

The **base** relation is the relation being versioned. The **v_added** relation stores the net tuples added (this includes replaces of tuples in the base relation) to the version, while the **v_deleted** relation has a single attribute (DOID) which stores the OIDs of tuples from the **base** relation which have been "deleted" or "replaced" in the version. Together, the **v_added** and the **v_deleted** relations form the delta relation of a version.

Forward deltas as we implement it allow for two alternative update semantics:

(1) Versions which can "see" changes made in the base relation; that is, the changes are propagated. We will describe in detail the implementation of this case in the next few sections.

(2) Versions which are snapshots of a base relation at a point in time; In this case, any changes to the base relation after that point in time will not be reflected in the version. This is implemented with rules similar to the rules used to implement the case above, except that the base relation is qualified by a time range (i.e. base["Jan 12 1990"] instead of just base).

The update semantics of both types of versions are such that the base relation is never modified by updates, deletes, or appends to the version.

3.1.1 Implementing Forward Deltas with Rules

Suppose that we have a relation called "base", and we want to create a version of it called "version". When we create a version, what is really created are the **version_added** and **version_deleted** relations. The rule for this is:

```
define rule v_create
on define version then do instead
 {
      retrieve into version_added ( base.all) where FALSE
      /* This query creates an empty relation with the same attributes as base */
      create version_deleted (DOID)

 }
```

As in Object Oriented Databases, we have a unique Object Identifier (OID) bound to each object/tuple in the database. Therefore, tuples in version_added and base which would otherwise be identical can be distinguished by their OIDs. The following four rules specify the semantics of versions :

APPEND :

```
define rule v_append
on append to version then do instead
append version_added (f1 = NEW.f1, ...)

/* NEW refers to the tuple-values being added to the version relation */
```

DELETE :

```
define rule v_delete
on delete to version then  do instead
 {
   delete version_added
        where CURRENT.OID = version_added.OID

   append version_deleted (DOID = base.OID)
        where CURRENT.OID = base.OID
 }

/* CURRENT refers to the tuples being deleted from version */
```

RETRIEVE :

```
define rule v_retrieve
on retrieve to version then do instead
retrieve (version_added | base).all
  where base.OID not_in version_deleted.DOID
```

REPLACE :

```
define rule v_replace
on replace to version then do instead
 {
   replace version_added  (f1 = NEW.f1, ...)
         where CURRENT.OID = version_added.OID

   append version_deleted (DOID = base.OID)
        where CURRENT.OID = base.OID

   /* The next rule is needed to append the
    * the base tuple to the version_added relation
    * if this is the first time the base tuple is
    * updated in the version.
    */

   append version_added (f1 = NEW.f1, ...)
        where CURRENT.OID not_in version_added.OID
        and CURRENT.OID = base.OID
 }
```

These four rules enforce the semantics of a version as:

version = (base - net_deleted_from_base) + net_added

which looks semantically similar to [STON 80, STON 81]:

version = (base U total_added) - total_deleted

It however does not have the deficiency of being unable to insert tuples that are identical to those previously deleted. This is because Postgres, being a relational/object-oriented system [STON 90] provides a unique object identifier (OID) for each tuple/object in the database. In addition, the "+" operator describes a concatenation instead of a relational union, because the intersection of version_added and version_deleted is empty. Thus, a retrieve on a version results in a scan on the version_added relation, and a two-way join on the base and version_deleted relations.

3.1.2 Query Processing for Forward Deltas

Given the set of rules shown in section 3.1, and the data shown in figure 2, we shall now illustrate how queries on versions get processed.

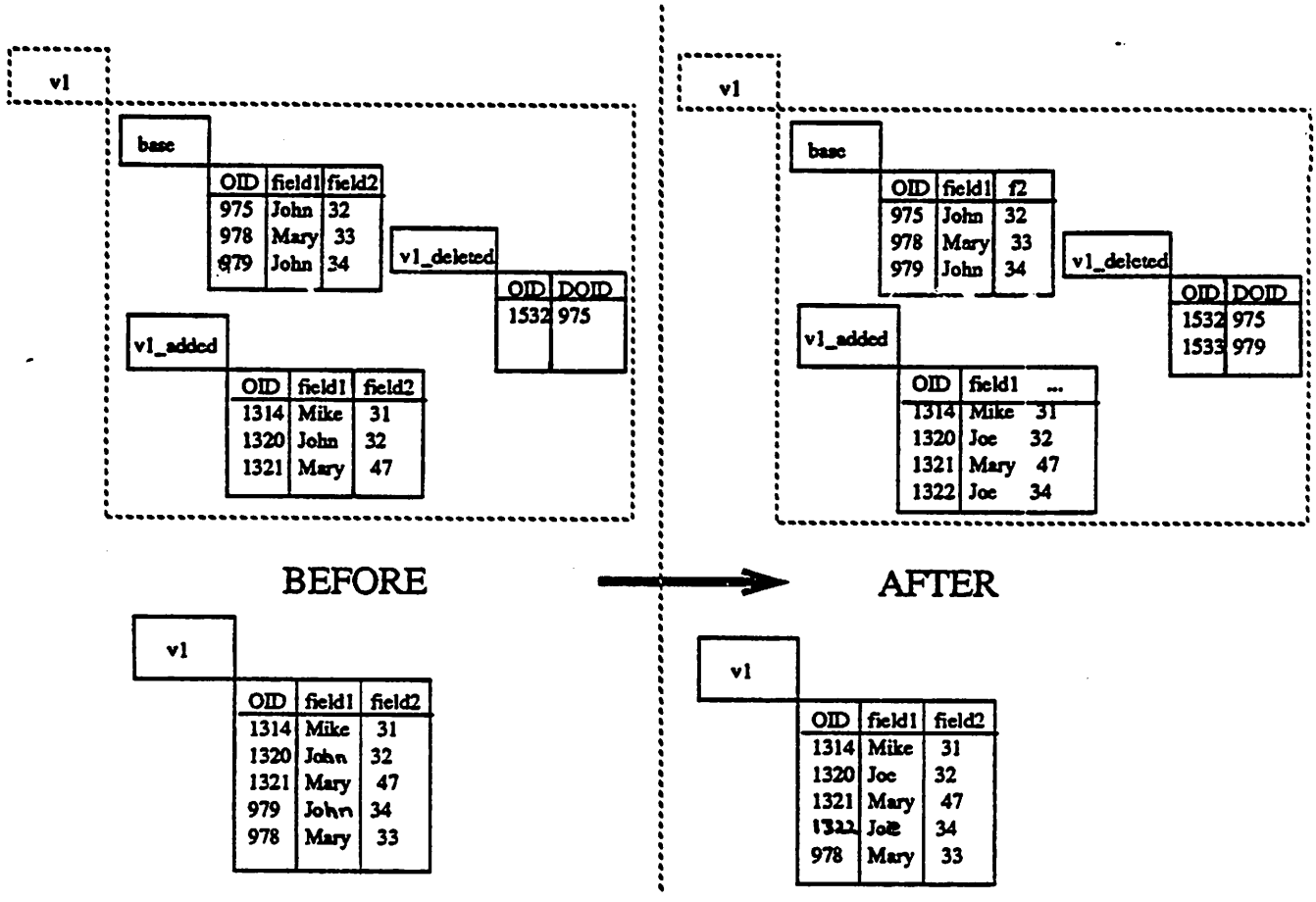


Figure 2: v1 is a version of base.

Now suppose we want to rename everybody named John to Joe. The query will be:

```
replace v1 (name = "joe")
  where v1.name = "john"
```

In PRS2, two things happens when the above query is executed:

(1) Since v1 is a version and has a set of rules defined on it as described in the previous section, the above query, when executed, will trigger the replace rule for v1.

(2) The semantics for the PRS2 rules is such that if a relation with rules defined on it appear in the qualification of a query, the retrieve rule for that relation is triggered. In the example, since v1 also appears in the qualification, the retrieve rule for v1is triggered. Hence the query will be rewritten as :

```
replace v1_added (name = "joe")
      where {v1_added | base}.name = "john"
      and base.OID not_in v1_deleted.DOID
```

```
        and {v1_added | base}.OID = v1_added.OID

    append v1_deleted(DOID = base.OID)
        where {v1_added | base}.name = "john"
        and base.OID not_in v1_deletd.DOID
        and {v1_added | base}.OID = base.OID

    append added(name = "goh", salary = {v1_added | base }.salary, ...)
        where {v1_added | base}.name = "john"
        and {v1_added | base}.OID not_in v1_added.OID
        and base.OID not_in v1_deleted.DOID
        and {v1_added | base}.OID = base.OID
```

After undergoing semantic optimization which removes redundant clauses, the query becomes:

```
    replace v1_added (name = "joe")
      where v1_added.name = "john"

    append v1_deleted(DOID = base.OID)
      where base.name = "john"
      and base.OID not_in v1_deleted.DOID

    append v1_added(name = "joe", salary = base.salary, ... )
      where base.name = "john"
      and v1_added.OID not_in v1_added.OID
      and base.OID not_in v1_deleted.DOID
```

Thus three things happens on an update to a version:

1 All qualifying tuples that are in the v1_added relation are updated

2 Qualifying tuples in the base relation which are not in the v1_deleted relation are "invalidated" by appending them to the v1_deleted relation.

3 If this is the first time a base tuple is updated in the version, it will be appended to the v1_added relation.

The net effect of these three queries will be to replace all qualifying tuples in the version. Given that subsequent retrieves on the version will retrieve tuples from

(added + (base - deleted)),

the replace query is correctly processed.

3.1.3 Cascaded Versions.

We can also create versions of versions using the rules stated in section 3.1.1 When we create a version of v1 named v2, v2 looks like :

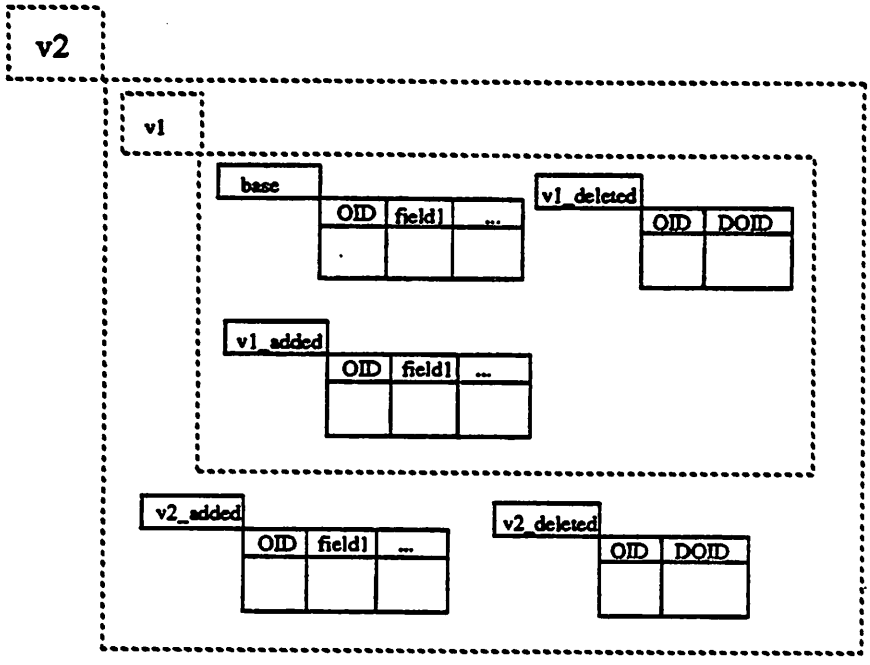


Figure 3: a version "'v2" of a version "v1".

Now suppose we do a retrieve on version2 using the following query:

retrieve (v2.name) where v2.salary < 5000

This query will trigger the retrieve rule for v2 and be transformed to :

retrieve ({ v2_added | v1}.name) where
{v2_added | v1}.salary < 5000
and v1.OID not_in v2_del.DOID

Since v1 is itself a version, the retrieve rule for v1 will be triggered, and the query gets rewritten to :

retrieve ({v2_added | v1_added | base}.name) where
{v2_added | v1_added | base}.salary < 5000
and {v1_added | base}.OID not_in v2_del.DOID
and base.OID not_in v1_del.DOID

This union query will be translated into retrieves on the individual relations v2_added, v1_added, and base. The net result of evaluating these 3 queries will be to a retrieval of all tuples in v2 which satisfy the user qualification. Thus, we have achieved the desired retrieval semantics for cascaded versions.

3.1.4 Comparing Efficiency

In this section we will compare the efficiency of our approach with that of [WOOD 83] since it is representative of relational systems that have added "special versioning enhancements". The algorithm for [WOOD83] is

(base U total_added) - total_deleted

while we use the algebraically equivalent

(base - net_deleted_from_base) + net_added

NOTE: "+" is the concatenation operator

that is, deleting tuples added since the version was created results in actually deleting the tuples from the added relation (this leads to no loss in generality since Postgres doesn't do deletions or replaces in place because of support for temporal queries).

Clearly, by using the net updates instead of the total updates, fewer tuples need to be processed, and as it turns out, the complexity of the resulting queries will also be less: Woodfill's algorithm results in two 2-way joins (base x deleted) and (added x deleted) , while ours results in one 2 way join (base x deleted) and a single scan on added) Thus, the algorithm we have presented in this paper should on average run slightly better than that presented by Woodfill. After an arbitrary number of versions, the difference grows even greater.

Let n = version number we are interested in
a = avg # of tuples added per version
d = avg # of tuples deleted per version
b = number of tuples in base relation
k = avg fraction of tuples changed between versions

Woodfill's algorithm runs in:

(n+1 + $sum from i=1 to n$ i) * (a + d)/2 + b

and the algorithm presented runs in:

((n - 1) * a) + ($sum from i=1 to n$ i) * (d / k) + b

For versions which are alternates, the dominating factor is k, since in the worst case all the tuples between successive alternates may be different. For versions which are derivatives, the level of nesting, "n", is the dominating factor. In both cases, the second algorithm runs faster in general.

3.2 Implementing Backward Deltas with Rules

As pointed out in [KATZ82], versions implemented using forward deltas necessarily results in somewhat inefficient access to the most recent version which must be painstakingly reconstructed from the base and the forward deltas from intervening versions. This is an especially crippling deficiency if the version system is being used to manage a design database or a CASE data dictionary where typically, the "current" or "working" version is often much closer to the most recent version (if not the most recent version itself). Thus, as suggested in [KATZ82], we can implement "backward deltas" or what Katz calls "negative" differential files.

In the versioning scheme using backward deltas, changes to the base are made in-place and the old values are recorded in the deltas. The old version (base) is defined by the new version and the differential files in [KATZ 82] as:

base = (new_version U total_deleted) - total_inserted.

Versions via backward deltas can be implemented by the following rules:

VERSION CREATION:

```
define rule v_create
on define version v1 of base
then do instead
 {
      rename base v1

      retrieve into base (v1.all) where FALSE
      /* create the relation with the same
       * attributes as v1, but with no tuples
       * in it, since rules trigger on access to
       * some "real" relation
       */
 }
```

RETRIEVE:

```
define rule v_retrieve
on retrieve to base then do instead
  retrieve v1["time at which this version is defined"]
```

The first command renames the current relation to be "v1", on which all updates and queries on the new version will be performed. The second command creates a relation (one with attributes and relevant structural information but physically containing no tuples). This is necessary to preserve the constraint that rules can only be defined on existing relations. Since Postgres provides support for such temporal queries and automatic access to archived or non-current tuples, the above renaming and rule

definition implements the semantics for derivatives. Subsequent versions perform similar renamings and rule definitions. Like most revision systems (RCS), updates can no longer be performed on an "old" version, although older versions may be materialized. Materializing older versions is easily accomplished by the following query:

```
retrieve into user_relation (version_we_are_interested_in.all)
        /* user_relation contains all tuples/objects
         * that would be in the version we wanted to
         * "check-out"
         */
```

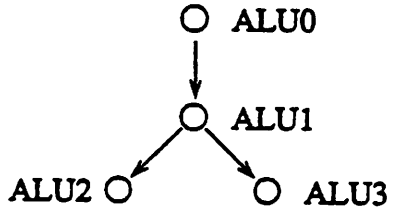
which will be appropriately transformed to the desired time-range query.

4.0 VERSION HISTORY AND VERSION GRAPHS

Version history keeps track of the is-derived-from or ancestor/descendent relationship between versions. In the simplest case, a version history is linear, but in general they are directed graphs. These graphs are easily modeled as relations that are updated on each creation of a version. To do this, we just add the following command to the set of commands executed at version creation time.

```
append (version_history.name = version.name,
        version_history.is_derived_from = base.name )
```

Thus, given the version graph of an object and its corresponding relational representation shown in figure 4:



The Version Graph

history

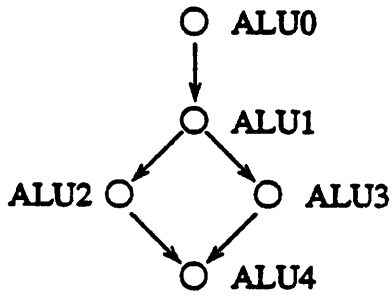
name	is_derived_from
ALU0	*initial*
ALU1	ALU0
ALU2	ALU1

The Version History relation

Figure 4: The ALU version graph before defining a new version of ALU.

If we define a new version "ALU4" which is derived from both ALU2 and ALU3, the tuples {ALU4,ALU2} and {ALU4,ALU3} would be appended to the history and the

resulting graph and relational representation would look like:



The Version Graph

history

name	is_derived_from
ALU0	*initial*
A:LU1	ALU0
ALU2	ALU1
ALU3	ALU1
ALU4	ALU2
ALU4	ALU3

The Version History relation

Figure 5: The ALU version graph after definining version 4.

5.0 EQUIVALENCES

"Equivalences" is the term used in [KATZ 88] to describe the multiple facets of a CAD object. For example, a two-input AND gate has three equivalent objects: functional description (Q = A ^ B), a structural description (the physical layout), and an extraction (the properties of the gate given the physical layout). There are two ways in which equivalent objects can be modeled in a relational database system. The first way is to treat equivalent objects as nothing more than different views of a single physical object. In this case, equivalences are really just different views of a relation, and versions of equivalences are just versions of views. Since these views are defined on a single relation, view updates are easily handled. The second way is to represent each equivalent object as a separate physical relation. In both cases, the database system needs to maintain consistency among equivalent objects. This will be discussed in the following section.

6.0 CONSTRAINT PROPAGATION

The different views (equivalences) of a CAD object must satisfy certain equivalence constraints. For example, a two-input AND gate has a functional description (Q = A ^ B) and a structural description (the physical layout), and an extraction (the properties of the gate given the physical layout). The equivalence constraint is that all three views

must represent the same object, in this case, a two-input AND gate. While changes to the physical layout of the AND gate do not affect its functional view, they do affect its extraction; that is, the properties of the gate (capacitance, resistance, speed, etc) varies according to its physical layout. Thus changes to the gate's physical layout must be automatically propagated to its extraction. This is called constraint propagation. We will now describe how constraint propagation can be implemented using rules.

If equivalent objects are treated as different view of a single physical relation where each attribute represents a property of the gate, constraint propagation can be implemented by simply defining rules on the relation. Therefore, some rules that will be defined in the above example are:

```
define rule rule1
on replace to AND.wirelen then do
replace CURRENT (resistance = compute_resistance(CURRENT.wirelen))

define rule rule2
on replace to AND.transistor then do
replace CURRENT (speed = compute_speed(CURRENT.transistor))
```

On the other hand, if equivalent objects are implemented as separate physical relations, then the rules need only be modified to:

```
define rule rule1
on replace to phy_layout.wirelen then do
replace extraction (resistance = compute_resistance(CURRENT.wirelen))

define rule rule2
on replace to phy_layout.transistor then do
replace extraction (speed = compute_speed(CURRENT.transistor))
```

In both cases, consistency is maintained among the different equivalent objects. As noted in [BATO 85], it is desirable to for new versions to "inherit" constraints of earlier versions. Thus we create new versions of an object or its equivalent objects, we must make sure that these constraints are regenerated for the versions. Since constraints in Postgres are defined using rules, we need only make sure that all rules defined on a base relation are propagated to its versions.

7.0 CHANGE NOTIFICATION

Instead of having users check if they have the latest changes, we would also like the version server to tell them whenever someone else has changed an object they are interested in. This is called change notification. Change notification can either be active or passive. In general, active change notification systems are message-based, and sends interested parties an electronic mail message when the object is changed, while passive systems are flag-based, and only puts a flag with the changed object so that the next time

the user browses through the list of objects, he can see which objects have been changed. Either of these systems can be implemented using rules. For example, the active system might have a rule that looks like :

```
on replace to foo
do execute(email("mike","foo was changed"))
```

in the above example, "email" is a user-defined Postquel function that sends an electronic mail message to a specificed user with a specified message. Execute is a Postgres command that is used to execute Postquel functions. The passive system, on the other hand, might have a rule that looks like:

```
on retrieve to list-of-objects
do replace (current.newchange = true )
where current.tmin > "last-time-I-browsed-this-list"
```

The passive system described in this section is similar to the timestamp mechanism proposed in [BATO 85].

8.0 CHANGE PROPAGATION

Change Propagation is a new, but relatively simple concept: It is the process where changes to a sub-component are automatically incorporated into composite objects that use the sub-component. For example, a register cell may be used in both the register file and instruction cache of a processor, thus changes to the register cell needs to be propagated to both the register file and the instruction cache. The rule needed to implement such a function would be:

```
define rule propagate-reg-cell
on replace to register-cell then do
 {

      replace instruction-cache ( cell = NEW.contents )
            where cell.name = "register-cell"

      replace register-file ( cell = NEW.contents )
            where cell.name = "register-cell"

      /*  and anything else that depends on register-cell */
 }
```

The rule given above does not automatically create new versions of the register file and instruction cache every time the register cell changes. In this case, the user must explicitly create a version if he wants the changes to be "frozen". Some version systems ([LAND 86], [KATZ 88], and [BEEC 88]) however, provide change propagation by automatically creating a new version of an object whenever changes occur in its sub-

component. This semantic can be supported by our framework with just minor variations to the update rule defined above. If we want to create a new version of the register file and instruction cache every time its register cell is changed, we simply modify the above rule to:

```
define rule propagate-reg-cell
on replace to register-cell then do
 {
      create version instruction-cache1 of instruction-cache
      replace instruction-cache1 ( cell = NEW.contents )
            where cell.name = "register-cell"

      create version register-file1 of register-file
      replace register-file1 ( cell = NEW.contents )
            where cell.name = "register-cell"

      /* and anything else that depends on register-cell */
 }
```

9.0 LIMITING THE SCOPE OF CHANGE PROPAGATION

While we would normally want changes to be propagated for code compilation, there are times in design databases when we do not necessarily want **all** objects that use a component be modified when that component changes. This is especially true if there are multiple paths in the object hierarchy. For example, although a register cell may be used in both the register file and instruction cache of a processor, we might want a new version of the cell that is faster but bigger to be used only in the instruction cache, where the improved speed is critical. Such semantics can be easily enforced by production rules. The example can thus be implemented by the adding the following rules:

```
define rule fast-icache
on replace to instruction-cache
  where NEW.speed < CURRENT.speed
then do instead nothing
      /* refuse updates if it slows the cache */

define rule small-regfile
on replace to register-file
  where NEW.space > CURRENT.space
then do instead nothing
      /* refuse updates if it increases the size */
```

The flexibility of the rules system allows for a finer degree of control by the user than is afforded by existing systems. If a user changes his mind about the update semantics, he can simply change the rules. In addition, by limiting the scope of change propagation,

the user effectively disambiguates the paths. Clearly, the generality of this method subsumes all proposed systems to date.

10.0 WORKSPACES

The idea behind workspaces ([KATZ 87], [CHOU 86]) is to allow users to each have their own little world. Workspaces are sometimes also called databases [KETA 87], federations [ECKL 87], and environments [SUN 88]. Most version models support three types of workspaces: private, group/project, and public/archive. Private workspaces are used exclusively by the user who owns it. Group workspaces may have several users updating it. Archive workspaces or Release versions contain only those designs which have been verified. Users can access the archive workspace only through a well-defined check-in/check-out procedure. Thus, the basic concept of workspaces reduces to a protection/permission scheme. In our framework, workspaces can be modeled by having a separate database per user, with a separate database that is owned by a "user" called archive. The only operations that are allowed for the archive database are "check-in" and "check-out". The check-in procedure typically contains a verification process to ensure that only consistent design objects exist in the archival database. Group sharing of objects amounts to having shared relations, and setting the permission to allow access to these relations to users belonging to the group. Two issues need to be resolved in order for us to effectively model workspaces using databases. The first is migration of data across different databases, while the second is the protection mechanism that needs to be implemented.

To support the migration of data across different databases, the database system must be able to support a hierarchical name space. With a hierarchical name space, commands like check-in and check-out can be implemented in the application program as the following set of queries:

```
checkin()
{
 Begin_transaction();
 verification-checks();
 define version x of /archive/y
 replace x (all = /username/foo.all)
   where any fields changed
 End_transaction();
}

checkout()
{
 Begin_transaction();
 retrieve into /username/foo
    /archive/foo.all
 End_transaction();
}
```

In general, both the checkin and checkout command can be precompiled for additional speed.

Since rules in Postgres are already used to regulate user access to various relations, it is trivial to extend the rules to provide the protection mechanism needed to support workspaces. Thus given the rule:

```
on {retrieve,replace,...} to shared_version1
where user.usename != "bob"
then do instead
  {/* nothing */ }
```

we can change it to:

```
on {retrieve,replace ...} to shared_version1
where user.usename != "bob"
and user.usename != "jhingran" then do instead
  {/* nothing */ }
```

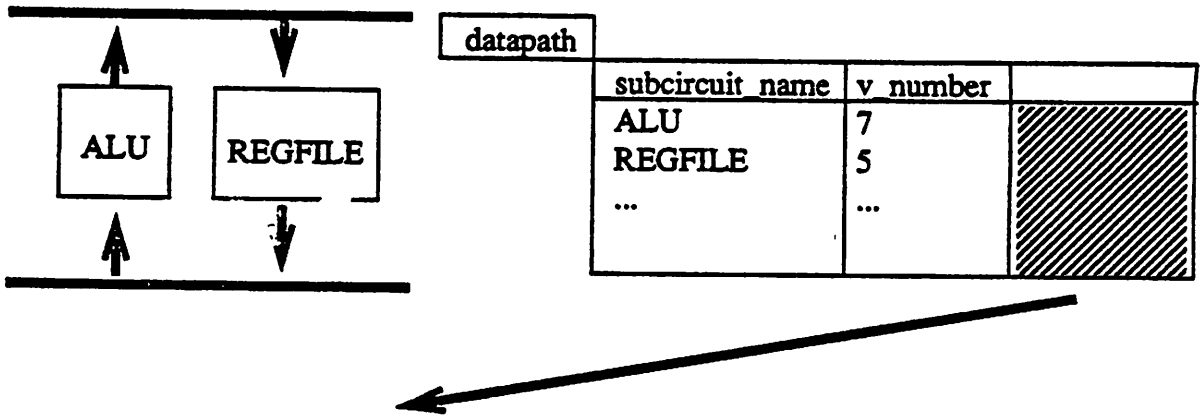
so that arbitrary access lists may be defined on any relation. Since views and versions are just virtual or partially materialized relations, these security mechanisms work just fine here too.

11.0 CONFIGURATION

Merging the concepts of version histories and component hierarchies results in configurations. A configuration is a composite object made up of versions of its sub-components which themselves might be composite objects. A configuration can either be flat (value-based representation) or hierarchical (procedural representation). In the former, the object is simply stored as a flat relation, while in the latter, the sub-objects are stored elsewhere and some reference (procedure) is kept with the object, thus preserving the hierarchy. In Postgres, sub-components are stored as procedural fields. To manage configurations, the system must be able to access the specific versions of the sub-components that make up the configuration, and maintain compatibility among sub-components of the configuration. Configuration can either be static or dynamic, and we will show how both can be implemented using rules.

For static configurations, all references to sub-objects in the configuration we are versioning must be fixed at the time we create a new version of the configuration. This is done by defining rules that transform accesses to sub-objects into references to the current "working" versions of the sub-objects. For example, the design shown in figure 6 is a configuration made up of two sub-components, each of which is itself a configuration.

Furthermore, let's suppose that the design uses version 7 of the ALU and version 5 of the register file, while the latest versions of the ALU and the register file are version 10 and version 8 respectively.




```
/* the procedure for the subcircuit as defined in rules */
define rule subcircuit
on retrieve to datapath.subcircuit then do instead
execute subcircuit(CURRENT.subcircuit_name,CURRENT.v_number)
```

Figure 6 : A configuration and its relational representation.

Since the design is a static configuration, the rules that are defined are:

```
define rule retrieve_alu
on retrieve to ALU then do instead
  retrieve ALU[7]

define rule retrieve_regfile
on retrieve to regfile then do instead
  retrieve regfile[5]
```

Thus the configuration (or its version) is "fixed" at the time it is created, and will not be affected by changes (or versions) to its sub-components.

For dynamic configurations, references to some sub-objects are amended only at the time the sub-objects are dereferenced. Therefore if the design in figure 6 is a dynamic configuration, then whenever the configuration is accessed, the version server would use the "working" version (not necessarily latest) version of each sub-component. It is up to the user to define the meaning of "working" versions. We note that the dynamic configuration is achieved by defining rules similar to those of the static configuration. Thus, if the design in figure 6 is a dynamic configuration, the rules that would be:

```
define rule retrieve_alu
on retrieve to ALU then do instead
  retrieve ALU["working_version"]

define rule retrieve_regfile
on retrieve to regfile then do instead
  retrieve regfile["working_version"]
```

This method is similar to the concept of parametrized versions described in [BATO 85].

12.0 CONCLUSION

In this paper, we have shown that production rules in a database system can provide a framework for version modelling that can be used to implement the needs of any versioning environment. A version system built from production rules not only provides all the capabilities of conventional version systems, but also has the advantage of being optimizable and easily extensible.

A major goal of this paper is not to adopt any particular version model. Thus in our discussion on the implementation details of various version mechanisms, we have tried to generalize the implementation of these concepts. As far as possible, we try to show how different semantics can be implemented with just minor changes to the rules. Throughout the paper, we have discussed our implementation of versions using names, but it would be trivial to modify the rules to use version-numbers. Finally, by making the version server a high-level system, we provide the users with a logical and intuitive framework in which they can represent and manipulate their favorite version semantics.

References:

[BATO85] Batory,D., W. Kim,
"Modeling Concepts for VLSI CAD Objects," ACM Trans. on Database Systems,
V. 10, N.3, (September 1985)

[BEEC88] Beech, D., B. Mahbod,
"Generalized Version Control in in Object Oriented Database,"
Proc. IEEE Data Engineering Conference, Los Angeles, CA, (Feburary 1988)

[CHOU86] Chou H., W. Kim, "A Unifying Framework for Version Control in aCAD Environment,"
Proc. 12th VLDB Conference, Kyoto, Japan, (August 1986)

[ECKL87] Eckland, D.J., E.F. Eckland, R.O. Eifrig, F.M. Tonge,
"DVSS : A Distributed Version Storage Server for CAD
Applications". Proc. 13th VLDB Conference, Brighton, England, (September 1987)

[JHIN88] Jhingran, A., M.R. Stonebraker,
"Alternatives in Complex Object Representation: A performance perspective,"
Computer Science Tech. Report, UCB?ERL M89/18, U.C. Berkeley,
(February 1989)

[KATZ82] Katz, R.H. and Lehman, T.J.,
"Storage Structures for Versions and Alternatives"
Computer Sciences Tech Report #479, CSD, U. Wisconsin-Madison
(July 1982)

[KATZ87] Katz, R.H., R. Bhateja, E. Chang, D. Gedye, V. Trijanto
"Design Version Management", IEEE Design and Test, V 4, N 1
(February 1987)

[KATZ88] Katz R. H.,
"Towards a Unified Framework for Version Modelling,"
EECS Division Tech. Report, UCB/CSD 88/484, U.C. Berkeley
(December 1988)

[KETA87] Ketabchi, M. A., V. Berzins,
"Modelling and Managing CAD Databases," IEEE Computer Magazine,
(February 1987)

[LAND86] Landis, G. S.,
"Design Evolution and History in an Object-Oriented CAD/CAM Database,"
Proc. 31st COMPCON Conference, San Francisco, CA, (March 1986)

[SEVE76] Severence, D.G., G.M. Lohman,
"Differential Files: Their Application to the Maintainance of Large Databases,"
ACM Trans. on Database Systems,V1, N.3 (September 1976)

[STON80] Stonebraker, M.R. and Keller, K.
"Embedding Expert Knowledge and Hypothetical Data Bases
into a Data Base System". Proc. ACM SIGMOD Conference,
Santa Monica, California, (May 1980).

[STON81] Stonebraker, M.R., "Hypothetical Databases as Views"

Proc. ACM SIGMOD Conference, Ann Arbor, Michigan, (May 1981)

[STON86] Stonebraker, M., L. Rowe,
"Design of Postgres,"
Proc. ACM-SIGMOD Conference on Management of Data, 1986

[STON87] Stonebraker, M., et. al.,
"Extending the Database System with Procedures,"
ACM Transs. on Database Systems,, September 1987

[STON90] Stonebraker, M.R., A. Jhingran, J. Goh, S. Potiamanos,
"On Rules, Procedures, Caching, and Views in Database Systems,"
to appear in ACM-SIGMOD Conference on Management of Data, 1990

[SUN88] SUN Microsystems,
"Introduction to the NSE,"
Part No. 800-2362-1300, (March 7, 1988)

[WOOD83] Woodfill, J., M. R. Stonebraker,
"An implementation of Hypothetical Relations,"
Computer Science Tech. Report, UCB/ERL M83/2, U.C. Berkeley
(January 1983)