Maintaining Integrity Constraints across Versions in a Database

Claudia Bauzer Medeiros
Geneviève Jomier *Wojciech Cellary*

Relatório Técnico DCC–08/92

Novembro de 1992

Maintaining Integrity Constraints across Versions in a Database

Claudia Bauzer Medeiros* Geneviève Jomier †
Wojciech Cellary ‡

Abstract

This paper analyzes the problem of maintaining application-dependent integrity constraints in databases for design environments. Such environments are characterized by the need to support different types of interaction between integrity maintenance and version maintenance mechanisms. The paper describes these problems, and proposes a framework in which they can be treated homogeneously. We thus bridge the gap existing between research on constraint maintenance and on version control, which has so far posed several problems to researchers in these two areas.

1 Introduction

A design environment (e.g., CASE, CAD) must support parallel development of independent entities (e.g., program modules, or car parts) that are eventually integrated into a final product (e.g., a piece of software, or an engine). During this activity, entities are logically grouped into

*Research partially financed by grants FAPESP 91/2117-1, CNPq RHAE/INFO 46.0571/91.5 and CNPq 453176/91

†Université Paris IX -Dauphine, 1 pl du Mal. Lattre de Tassigny, Paris

‡Institute of Computing Science, Technical University of Poznan, 60-695 Poznan, Poland

sets whose components "go together", i.e., satisfy a certain unit specification. These sets are called *design surfaces*. Other common names are *configurations* or *contexts*. A surface usually corresponds to a design unit, assigned to a given design team.

In order to support the work of these design teams, the environment must provide the means for ensuring the integrity of the different surfaces, and of keeping track of their versions. It must also provide communication channels among the teams, and supply tools for progressive integration of the design surfaces. Managing the coordination of integrity and version control is therefore a crucial issue.

Designing involves an intensive work of experimentation over different alternatives, where entities are created, updated, destroyed and merged. In order to support these activities in a database environment, several version support mechanisms have been proposed (e.g., [KSW86, Kat90, CJ90, TG92, KS92]), and recently surveyed in [BBA91]. Very little is said, however, on the maintenance of integrity constraints in such a context.

Since the design activity is exploratory by nature, different versions of any given entity may exist simultaneously in a database. Controlling the database's coherence requires therefore maintaining versions of surfaces, i.e., finding out and keeping track of the appropriate component versions that form a consistent surface. The integrity problem acquires thus two dimensions: maintaining the consistency of a surface that has several components, and maintaining a component's integrity. These two dimensions are translated into two problems: maintaining different versions of components and surfaces, and integrating component versions into an appropriate surface version.

Due to the complexity of these issues, the integrity problem in a version context is usually limited to integrating component versions into a consistent surface version. Thus, the consistency concept in a version environment is mainly that of determining sets of consistent versions of data [KSW86]. When a new component version is created, the first goal is that of maintaining the entity's (local) consistency, followed by trying to integrate this version into some consistent view of the database.

As we will show, the consistency issue has other ramifications which so far have been ignored by researchers: constraint evolution during the design process; surface evolution; and interference among surfaces. The two latter involve issues similar to the ones treated in the view update problem (e.g., [BS81]).

This paper presents a framework that allows homogeneous treatment of versions and of constraints, based on the database version mechanism described in [CJK91], and extended in [CVJ91]. We show how this framework can be used to manage versions of constraints, and constraints which are imposed over the version generation process. The latter is a new type of constraint, which we call *version control* constraints. We follow the division proposed in [Sci91], and distinguish logical and physical versioning. We present all issues from a logical (design) level, rather than from a physical (implementation) level. For implementation details of the mechanism discussed, the reader is referred to [CJ90]. The main contributions of this paper are the following:

- Systematic discussion of the problems of supporting the interaction between integrity control mechanisms and version control mechanisms in a design environment.
- Presentation of a framework that allows treating these issues in design databases. This framework encompasses versioning of constraints, which has so far been ignored in the literature.
- Discussion of a new family of integrity constraints, *version control constraints*, which are rules that define legitimate ways of creating and maintaining relationships among versions.

The results presented are general, and independent of the data model. To stress that, we will refer to database *entities* instead of the term *objects*. The latter, in this paper, refers to entities in an object-oriented database, and which should be thus understood in the corresponding context. A database *entity* is a set of database elements that represent a design component and may for instance be a tuple or a relation in a relational database, or a set of objects in an object-oriented database.

The term *object*, on the other hand, is used here as an instance of a class in an object-oriented database. It is characterized by its state (contents) and behavior (methods), and is subject to inheritance and composition properties [Bee89].

This paper is organized as follows. Section 2 presents the terminology used and some issues concerning constraint maintenance and version generation mechanisms. Section 3 discusses the problems of maintaining versions in the presence of integrity constraints, and presents the approach of [CJ90], which we extend to unify the concepts of version and constraint maintenance. Section 4 presents two new problems - versioning of constraints and constraints over version generation - and shows how our approach can be used to solve these issues. Finally, Section 5 contains conclusions and directions for future work.

2 Research on versions and on constraints

An *integrity constraint*, in a database environment, is a statement of a condition that must be met in order to maintain data consistency [JMSS90]. The problem of automatic maintenance of integrity constraints in a database management system (DBMS) has been extensively considered in the literature. Since database systems have limited support for such a facility, application developers are forced to embed code in each application, in order to verify constraints. This has also the inconvenient of making all applications very sensitive to any modification in the constraint set, besides leaving to programmers the burden of having to know and check all relevant constraints at each step.

The issue of database *versions* is another problem that has been intensively researched in recent years. A version is a "semantically meaningful snapshot of an object at a point in time" ([Kat90]). Among the problems that have to be solved in version management are the maintenance of historical data, the sharing of versions, and the control of concurrency for version access.

Research on constraints and on versions have for the most part been conducted independently. Version management mechanisms implicitly

assume that existing integrity constraints are somehow maintained when a version is created. In other words, no entity version may reflect an inconsistent state of the world. By the same token, the research on constraint maintenance does not take the possibility of versions into consideration. It only addresses the issue of ensuring global consistency of a database, given a set of updates. In a version environment, however, this is not always feasible.

The main issue is that, once we are faced with a world with versions, we lose the traditional notion of transaction correctness which underlies the standard constraint maintenance paradigm. A *transaction* is a sequence of operations that takes a consistent database state into another consistent state. A transaction under the version paradigm does not obey this definition, since the notion of a globally consistent database no longer holds. Thus, it has been impossible to integrate results of research in integrity management and version management since each forces the adoption of a different consistency definition paradigm.

2.1 Issues on constraint maintenance mechanisms

The development of efficient mechanisms for performing maintenance of integrity constraints is still a difficult problem. Such tasks are mostly left to database designers or application programmers. Designers try to ensure integrity by following specific design rules, and defining protection schemes for sensitive data. Programmers have to encode integrity maintenance into applications, which is very costly, since it means modifying code whenever the integrity characteristics of a system change. A partial solution offered to this problem is that of maintaining constraints by means of production rules, using the active database paradigm (e.g., [KDM88, DHL90, GJ91, BM91]).

Integrity constraints may be application-dependent (defined by the user) or model-dependent (and thus ensured by the DBMS). We will not worry about the latter, since we assume they are ensured by the underlying database management system. What concerns us is the *maintenance of application-dependent constraints in a version environment.*

Constraints can be classified in several ways. One of these many classifications relies on distinguishing *static* constraints – i.e., those that define a valid database state and are usually stated using first order logic – and *dynamic* constraints – that specify valid database state transitions, and are specified using modal logic. One important set of dynamic constraints is that of *temporal* constraints, that determine coherent sequences of state transitions over time [JMSS90].

Though the notion of constraint is usually associated with modification of data, the database schema composition (i.e, the specification of the database components and their types) may also be subject to restrictions. One can, for instance, add new fields to a relation, or change the composition graph of a class in an object-oriented database.

In an object-oriented world, constraints can be defined not only over structure or contents, but also over *behavior*. As pointed out in [MP91], these constraints correspond to establishing consistency directives for methods. We take the view that behavior description is part of a database schema. Static constraints on methods are, for instance, domain constraints over their parameters. Dynamic constraints over methods are, for instance, the specification of allowable sequences of method execution. Thus, we consider that the constraints that interest us can be either static or dynamic constraints, and may apply to the data or to the schema of a database, where the schema, in object-oriented databases, includes behavior specification.

2.2 Issues in version mechanisms

Versions are present in all stages of a design process. As remarked in [EL90], it is impossible to give a formal specification of a version. The task of determining what constitutes a version of an entity is left to the user.

Normally, there is a distinction between two types of version creation operations: *revisions* and *alternatives* (also called *variants*). A *revision* corresponds to an improvement; its creation is thus always dependent on a predecessor. A typical use of revisions is found in the historical

approach to versions, which considers the evolution of entities through time, and only the latest version is updated [Bla91]. Once a revision is established, none of its previous versions in time can be changed. *Alternatives* correspond to variations in a given version, in response to different prerequisites (e.g., performance) but which are functionally similar. CAD applications use alternatives extensively, by considering the parallel evolution of several versions of an entity [BBA91]. It is up to the user to determine whether a newly created version constitutes an alternative or a revision of an existing entity.

The version creation process for an entity is frequently depicted by means of different types of graphs, where each node corresponds to a version of the entity. The most common of these is a directed acyclic graph – *dag* – where a node's descendants along a path are its successive revisions. The edge from a node to its descendant indicates a temporal creation order. Nodes that have the same immediate parent constitute a set of alternatives. Revisions can be created from several parents at the same time, which corresponds to the merging (or consolidation) of concepts.

As mentioned in the introduction, designer teams work on a set of entities, that form a design unit – the surface. Maintaining the consistency of a surface requires correctly matching entity versions, i.e., a set of entity versions that form a coherent functional system unit. The design process in the presence of versions requires progressively merging different (consistent) surfaces until the final product is achieved. Since designers work over a surface in order to obtain a final product, most of the work in version support mechanisms is concerned with providing users with adequate surface management.

3 Maintaining constraints in the presence of versions

The consistency problem in the presence of versions has so far been considered as a problem of determining and maintaining consistent surfaces.

There are, however, several other issues to be considered:

1. **Constraint evolution**. First, during any design process the set of constraints is also subject to evolution. In fact, as stressed by [CVJ91], the notion of consistency evolves over time. Thus, one is now dealing with a world where constraints themselves are subject to modifications.

2. **"View" updates**. Second, the design process is often undertaken by a group of designers, and the project is divided in subtasks, where different groups design different parts for future integration. Each group sees a set of entities over which it has to configure and maintain the appropriate surfaces. However, maintaining local (surface) consistency is no longer sufficient, since decisions taken by a group may affect the decisions taken by a different design group that sees *another* portion of the database. Therefore, the usual local consistency model is not enough and we are faced with a variation of the view update problem, where the view is now a design surface.

3. **Propagation of constraints**. Finally, the evolution of constraints may occur simultaneously in several design groups. In some cases, constraints which have been added or modified by a group will have to be adopted by other groups. One thus sees the phenomenon of update propagation from a new prism: researchers limit themselves to analyzing effects of propagating updates on data, whereas in a design environment constraints themselves are propagated along different surfaces.

3.1 Evolution of versions under constraints

Figure 1 shows two graphs. The one on the left describes the version evolution of a surface S, and the one on the right the evolution of a design project to which this surface belongs. The first graph shows the (database) view of a designer team working on surface S. The surface

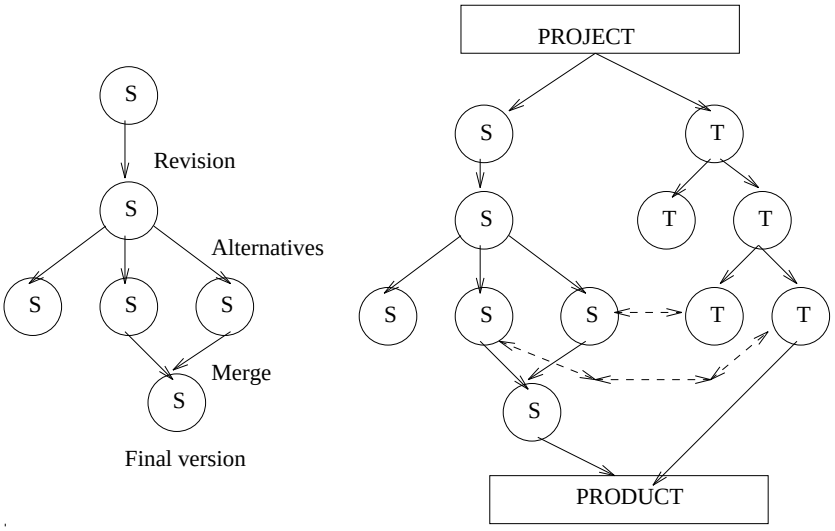

Figure 1: Version Graphs

goes through a set of revisions and alternatives that are merged into a final version. Some alternatives are abandoned.

The second graph shows the project manager's global view (in this case, surfaces S and T). After an initial (independent) design stage, the surfaces interact (shown by the dashed lines). This interaction is not necessarily visible to the design teams, and corresponds to constraints across surfaces visible only at the global level: design decisions taken for T affect the future evolution of S, and vice-versa, since the product needs eventually merging S and T.

Of course, this is just a simplified example, since in most cases surfaces have intersections from the start (e.g., in software design, where modules developed by different teams use the same data structures, or must communicate via a common interface). The figure, however, illustrates some of the problems of interaction of constraints and versions:

- global constraints - these are overall project constraints, which must be kept for all design stages of all surfaces (e.g., enterprise

price policy, or a global data model).

- local (component) constraints - these are constraints that apply to a single design entity (e.g., dimensions of a surface component).
- local (surface) constraints - these are constraints that apply to a surface (e.g., to S), and which are ignored by other surfaces (e.g., functional specification).
- inter-surface constraints - these are constraints imposed across surfaces (e.g., interface definitions for future integration). These are the constraints that control the interaction between T and S.

Let us illustrate these issues using a simple example. Suppose that a project uses a relational database, where each surface is stored in a relation. A global constraint is enforcement of 1NF. Local entity constraints are those that apply to a given tuple, or attribute (e.g., domain). Local surface constraints apply to one relation alone (e.g., a functional dependency). Finally, inter-surface constraints would restrict sets of relations (e.g., referential integrity constraints). Coordination of versions in this system means maintaining all these dependencies for all versions of tuples, relations and attributes. However, multiple versions of any given relation, say R1, must coexist in the system. Each version of R1 is consistent with respect to its local constraints, and to the database's global constraints. On the other hand, the set of versions of R1 may violate local constraints (e.g., a functional dependency). By the same token, consider a constraint that determines valid relationships between R1 and another relation R2, which also has several versions. This constraint is not supposed to hold between any two versions of R1 and R2. Rather, it has to be maintained between appropriate pairs of versions of these relations.

The greatest difficulty in consolidating work in constraints and in versions lies in that researchers in these areas live in different worlds, and cater to different users. Integrity constraint maintenance presupposes that users have a comprehensive view of an application, and that, even

if the world evolves, the database will harmoniously follow this evolution, by changing state (e.g., schema or data updates). Version management presupposes that users are dealing with tentative specifications, working on different alternatives at the same time. In this world, evolution means not only changing the one state, but keeping different states alive and available at the same time, keeping track of them all for future modifications. Research and mechanisms for constraint maintenance are geared towards moving a database from one state to another. Research and mechanisms in a version environment concern moving a database from a set of states (the surfaces) to another set of states.

Thus, standard integrity maintenance mechanisms are useless in controlling integrity across sets of versions. In fact the traditional notion of constraint ceases to hold, once we introduce versions, since constraint maintenance mechanisms tacitly assume there is just one instance per entity, whereas version mechanisms assume there are several different instances for the same entity. The existence of a database with versions means therefore the existence of a database that is by definition inconsistent in the traditional sense. The consistency problem becomes therefore that of *finding surfaces*, or sets thereof (i.e., partial rather than global consistency).

We summarize the problem as follows:
Maintaining integrity constraints in the presence of versions is a problem of finding and maintaining surfaces across a database, where each surface should be kept consistent with respect to the constraints. It entails maintaining consistency within a single database with several entities, where to each entity may be associated several data versions, several schema versions and several (versioned) views.

The introduction of the object oriented paradigm increased the difficulty of dealing with versions in the presence of constraints. When a composite object is updated, one must determine how to propagate the update to the appropriate component versions, which is difficult when there are different alternatives for component and composite objects. The task is therefore that of finding first the appropriate surface for a composite object, and then of finding the schema version to which that

surface belongs. Thus, some researchers restrict themselves to versioning object schemas (e.g., [ALP91]) and others yet to the problem of versioning isolated objects while maintaining the schema invariant.

Suppose for instance we have sets of **S**eats, **W**heels, **B**rakes and **G**earboxes (components) which we want to use to design **Land Vehicles** (surfaces). Cars and Trucks both need **S**eats, **W**heels, **B**rakes and **G**earboxes, but under different versions. Depending on how we put them together, we will come up with a different product. Though each component is consistent with respect to its local constraints (e.g., a **W**heel should be round) the final product may be subject to a different set of constraints (e.g., the weight of the **Land Vehicles**). A different component version does not necessarily mean difference in stored data but different behavior. Thus, two versions of the same component may react differently to the same method (e.g., *fold* applied to **S**eat).

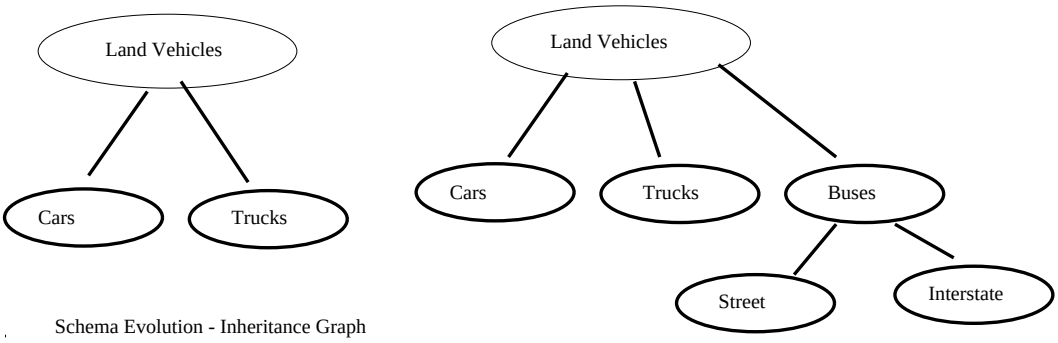

Figure 2: Schema evolution

During the design process, the schema may evolve for certain alternatives (see Figure 2). Thus, if we decide to design Buses, we will need different (and more) **S**eats and **W**heels. Thus, a given **S**eat version will only fit **Land Vehicle** versions of Buses. Interstate-Buses will furthermore need a new component - **S**anitary_**F**acilities.

3.2 Introducing the VBD framework

Until recently, all the issues discussed previously were considered under the assumption of a unique database, where surfaces had to be rebuilt at every new entity version, rending global consistency control practically impossible. The first attempt to relax this monolythical framework was the VDB model, presented in [CJ90], and extended and formalized in [CJK91], in the context of object oriented databases. In this model, instead of keeping track of versions of individual entities, the authors treat the problem from a point of view where a unit of versioning is the entire database, rather than parts thereof.

In the standard approach, when some entity version is created, several links have to be established, to determine to which consistent surface the version should belong. In the VBD approach, instead, the creation of an entity version entails the creation of a new logical database which will contain this entity, and all surfaces to which it belongs. The VBD approach sees therefore an augmented database that contains a set of logical databases, which differ from each other in the entity versions and the surface versions they encompass.

Creating a **S**eat version S1 for a Truck verson T1 means the creation of a logical database Dk, which contains one single seat version (S1) for a given truck version (T1). The user who is working with Dk ignores the existence of other Truck or seat versions. Dk is a database which ignores the notion of versions.

Informally, this model can be described as follows: "Let $\mathcal{D}$ be a VBD database. At any instant in time, the state of $\mathcal{D}$ is described by a set of logical database states { D1, D2 ... Dj ... Dn }, where each Di is consistent and any two elements Di, Dj differ from each other by at least one entity version. If E is some entity in some database Dj, the creation of a version E' for E logically corresponds to the creation of a *new consistent logical database* Dj' which contains E' instead of E. The new state of $\mathcal{D}$ is thus $\mathcal{D}$ U {Dj'}. Each Dj is a non-version VBD databse view.

The evolution of a VBD database in the presence of versions can be

seen as a sequence of two transactions, as follows:

1. the first transaction generates a new logical database, Dj', which is an exact copy of Dj. Since Dj was supposed to be consistent, then Dj' is also consistent;

2. the second transaction operates only over Dj', as if it were the only database in $\mathcal{D}$, and changes E to E'. Furthermore, it may modify other entities in Dj' in order to achieve consistency *within Dj' only* - in the same way that a transaction (in the standard nonversion model) will propagate updates in order to achieve a final consistent state. Thus, it takes Dj' from a consistent state to a consistent state.

The modification of other entities by the second transaction corresponds to finding the appropriate surface for E', creating in the process other entity versions, if necessary.

We like to compare the evolution of a VBD to the creation of database "slices " in time. Each slice is a logical database, which is consistent and can evolve independently. Creating a new version for an entity corresponds to the appearance of another slice. From now on, to help text understanding we will refer to a "slice" meaning "a logical database inside a VBD database". It must be stressed that slices in the VBD approach are not snapshots. Rather, they are databases that can be updated by the user. These updates do not in themselves force the creation of a new slice. It is up to the user to determine from a slice's state whether the sets of updates that have been applied to it configure the need for another slice. This is similar, up to a certain level, to the notion of a temporal database, which can be seen as a sequence of past database states (the temporal slices) along time (e.g., [Sno90]).

In a design environment, a given design group does not need to see a complete database, just the set of entities that correspond to the task the group is developing. The VBD model was extended in [CVJ91], for object-oriented databases, to take this into account. Each VBD slice is locally consistent, and a design team only works within a slice at a

time. A slice thus constitutes a database view for a team. We have thus fallen back into the usual transaction model, and need only worry about consistency within a logical database (a slice). Each slice is now a standard database without versions. It is is consistent in the traditional sense, and can be updated at will, independent of the other slices that exist at the same time in the VBD. One does not have, therefore, to deal with multiple versions of an object within a slice, since each version will belong to a different slice.

Consider again the Truck example. Let us consider three of its components: (**W**heel, **G**earbox, **B**rake). Each component is assigned to a design team that will develop it along different versions. Global constraints apply to every **Land Vehicle** (e.g., relationship between material used and maximum component weight). Consider initially a first state with two **W**heel alternatives (W_1 and W_2), one **G**earbox G_1 and two **B**rake alternatives (B_1 B_2). Supposing that we can make Trucks with any combination of brakes, wheels and gearboxes, we have a VBD with the following slices: {D1=(W_1, G_1, B_1), D2=(W_1 ,G_1 B_2), D3=(W_2 , G_1, B_1), D4=(W_2 ,G_1 B_2)}.

This database has four logically independent slices that may now evolve independently (and can be considered as consistent views of the VBD). Suppose now that, in the next design stage, **B**rake B_2 suffers a revision and becomes B_3 which will force updates to slices D2 and D4. Suppose that B_3 is not consistent with **W**heel version W_1. In this case, only D4 can be updated, becoming D5 (since D2 contains W_1) .

D2 should be abandoned in the design process, since its **W** component is not consistent with the revision of **B**. This is an example where an integrity constraint, combined with the version mechanism, forces discarding a slice. Thus, we now have {D1=(W_1 G_1 B_1), D3=(W_2 G_1 B_1), D5=(W_2 ,G_1 B_3)}.

If we now create an alternative to G_1 called G_2 which is consistent with W_1 but not with respect to W_2 we will have the following VBD {D1=(W_1 G_1 B_1), D3=(W_2 G_1 B_1), D5=(W_2 G_1 B_3), D6=(W_1 G_2 B_1)}. Notice that now no slice is abandoned, since the version process created an alternative (and thus allowed maintaining older versions of the

same component). The team responsible for surface W_1 will work independently over D1, D3 or D5, each with a different view of the database. Figure 3 shows the VBD evolution together with surface evolution, where the time axis is situated vertically.

4 Constraints in the VBD framework

Given the VBD framework, we can now analyze the constraint maintenance problem in a multiversion database as follows. A database with versions is a set of slices where each slice is a consistent database of its own - in fact, slices differ from each other by containing different versions of given surfaces. When we apply updates to a slice we need never worry about other slices in the system, and updating the slice does not necessarily imply the creation of a new slice[1].

The data model is immaterial - for instance, in an object-oriented world, update propagation along an object's components is a local problem that will be performed within the slice that contains the object, without caring about other versions of the same object that exist in other slices. By the same token, we do not have to worry about matching schema and data versions. These problems disappear since by definition all operations are applied within a slice.

So far, we have stated the following points:

- We solve the standard problem of constraint maintenance in the presence of versions (i.e., maintaining surface consistency) by adopting the VBD model, where these two issues can be treated orthogonally.

- In the VBD model, each surface version in a database belongs to a slice – an autonomous database. This slice is a database view for a design team. Thus, only a subset of the VBD's surfaces have

[1] This does not mean that traditional update problems have been solved. It only means that update control mechanisms do not have to be burdened with versioning at the same time.

non-null values in this slice. The remainder of the database is ignored by the design team. Consistency maintenance is restricted to maintaining integrity within a slice (and thus to its set of surfaces). This is another formulation of the view update problem.

- Local (i.e. slice) consistency implies global (i.e. VBD) consistency.

The second statement shows that we now face a standard view update problem. The third statement is the usual assumption in integrity maintenance research [JMSS90]. We point out that these assumptions underly all research in integrity maintenance in databases. We have thus transformed the problem into that of traditional (slice) integrity maintenance.

The above is sufficient for maintaining application-dependent integrity constraints in the presence of versions, in the traditional sense. We now extend the problem to show how we permit other types of constraints not considered in the literature.

4.1 Introducing versions of constraints

In a design environment, constraints themselves evolve as more knowledge is acquired about the end product. Thus, one should consider the situation in which constraints themselves are versioned: there may be scenarios where some constraints do not yet exist, and others in which they are not completely defined. Two different versions of an entity may exist not because the entity itself was directly subject to a change, but because the constraints to which it was subject were modified.

In the previous section, we showed how to solve the problem of maintaining immutable constraints for a given surface. To completely characterize the evolution of versions in a database, we must also consider versions of constraints. This problem has not yet been treated, to our knowledge, due to the difficulties it adds to a classical version environment. How can one establish connections among different entity versions, if a given entity may be subject to different constraints over time?

Again we find the answer to this problem in the VBD model. We recall that a given slice in a VBD is a database on its own. Thus, it obeys

the (local) constraints, which may happen to be a particular version of the constraints that apply to other slices in the VBD. Therefore, supporting versions of constraints becomes yet another case of slice generation. We treat integrity constraints as properties of a database schema, and thus maintaining versions of constraints is one instance of the more general case of maintaining versions of schema. The two-transaction model remains the same. First, one creates a new slice which is an exact copy of some other slice. Then, the constraints in this new slice are updated (deleted, changed, or introduced), and existing data is changed in order to adapt to the new local conditions.

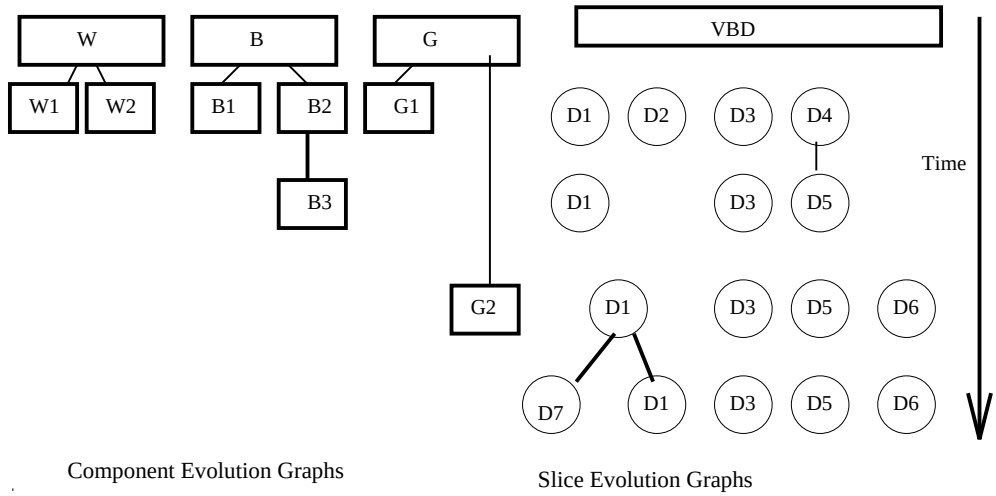

Figure 3: VBD evolution along time

Let us return again to the Truck problem, with 4 slices (D1, D3, D5, D6). Assume that the designers want to try an alternative constraint on the speed of Truck of slice D1, which is reflected on the form its **W**heels are connected with its **G**earboxes. We now have an alternative to D1, called D7, which obeys different (local) constraints which are not shared by the other slices. Notice no change was applied to the components, only to the way they are integrated into the surface.

The new VBD is thus {D7=($W_1\, G_1\, B_1$), D1=($W_1\, G_1\, B_1$), D3=($W_2\, G_1\, B_1$), D5=($W_2\, G_1\, B_2$), D6=($W_1\, G_2\, B_1$) }, where the local values of W_1 , G_1 and B_1 were maintained, but an alternative was necessary since new constraints apply to the slice. Being an alternative, slice D1 was not discarded.

4.2 Introducing constraints across versions

Research on constraints considers static and dynamic constraints over data and over schema. The VBD framework allows us to look at a multiversion database as if it were a set of independent databases, each of which is consistent. We know, however, that these databases are actually far from independent. In fact, their creation follows some order over time, and a new slice is obtained from some others by a set of operations: revisions, alternatives or merges.

When we superimpose the standard constraint classification (dynamic and static) and the VBD model, we see that besides the constraints imposed at the "local" (slice) level, users specify constraints on version generation and maintenance, which are applied over sets of slices, i.e., globally. We thus see the emergence of a new class of constraints, which establishes consistency relationships among slices, and which in the VBD model holds over a different dimension, i.e., *across* logical databases rather than *inside* a database. We claim that this global type of constraint is different in nature from other integrity constraints, and that only by treating the problem in the VDB framework this issue can be properly understood. Furthermore, this type of constraint has not been considered previously.

These new constraints are imposed on *version generation and maintenance*. We call them *version control* constraints, in that they consist of constraints that determine when and how to create new surface versions, and in what way and to what degree these versions must preserve consistency among themselves.

The version evolution of an entity is traditionally represented in a dag. We extend now this notion, and propose a database evolution dag

for the VDB model, where each node is a slice instead of a component version. In fact, we impose a global constraint to the slices within the VBD, forcing that they be related to each other according to a *slice generation dag*. A given path indicates chronological creation of revisions.

Version control constraints are also application-dependent. It is these constraints that determine when a revision or an alternative should be created or deleted. They can also be dynamic or static. A dynamic constraint controls *creation or deletion* of slices, based on information from the existing slices, therefore controlling changes in the graph topology. A static constraint across slices controls updates on a slice (without changing graph topology), based on information about the state of *other* slices. One has thus two dimensions over which constraints can operate: in the standard sense, dynamically or statically, inside a given slice, when it is updated without creating a new version. And dynamically or statically as regards version generation, as sets of versions are considered as a whole.

As an example of static version control constraints, suppose the project administrator decides that the **G**earbox constraints in D7 are the ones that must be obeyed by all future **G**earbox instances. Remember that D7 uses gear alternative G_2. This means that all future slice versions will have to use **G**earboxes that obey the constraints on G_2. Furthermore, every time G_2 constraints change, all surfaces will have to change. This is equivalent to saying that the constraint on the **G**earbox component must be obeyed by all **G**earbox versions, and also that this constraint must be obeyed by all future slice updates. Thus, a constraint on a slice component affects the future evolution of slices.

One example of dynamic cross-slice constraints are temporal constraints applied over a (historical) sequence of revisions (i.e., along a VBD dag path) - for instance, the cost of Truck versions can never decrease. When a path node is updated, a new slice - its revision - can be created only if this constraint is obeyed. A dynamic constraint over a single version is maintained locally (in a slice) when that version is updated, and is not connected with version generation. A dynamic constraint over a set of versions imposes conditions on the derivation of a

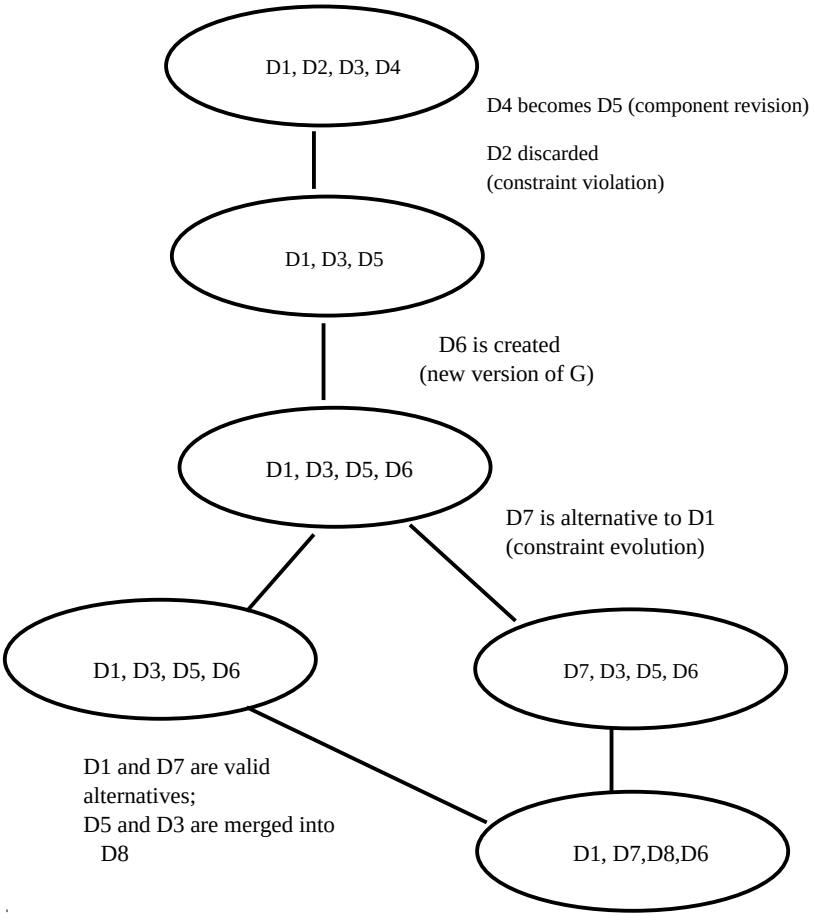

Figure 4: VBD evolution dag - Version control constraints

new version. We have thus established a new type of dynamic constraint: a constraint across slices in a VBD.

Another dynamic version control constraint may state, for instance, that it is possible to merge slices as long as they differ only on one component. Thus, D3 and D5 can be merged (since they differ only on **B**rake), but D3 and D7 cannot be merged (since they differ in **W**heel and **G**earboxes). Figure 4 shows the evolution of the VBD's set of slices, where the edges have been annotated with the reason for the transition. Suppose now that revisions of D7 and of D3 are created whereby they acquire the same **G**earbox component (say, G_5). These revisions will now be allowed to merge, since they will differ only in the **W**heel component.

5 Conclusions and directions for future work

This paper presented a new framework that allows considering the integration of version maintenance and integrity control mechanisms. We showed that by using this approach the two problems can be considered independently, and thus solved at the same time without the usual interference seen in other models. We also showed how the constraint maintenance issue in a multiversion environment can be treated at two different dimensions - locally, over one slice, and globally, across sets of slices.

We showed how in this framework one can treat uniformly several problems that have so far been ignored by other authors, such as the versioning of data and schema, and versions of constraints themselves. Finally, we introduced version control constraints, which is a new class of constraints that controls the process of legitimate version generation.

The VBD model is being implemented for the O2 database system. Among future directions of our work are the extension of [MP91] to that of maintaining integrity constraints in a multi-version object-oriented database, and the managing of temporal properties through versions.

References

[ALP91] J. Andany, M. Leonard, and C. Palisser. Gestion de l'Evolution du Schema d'une Base de Donnees. In *Proc. PRC - BD3*, pages 7–24, 1991.

[BBA91] M. Borhani, J-P Barthès, and P. Anota. Versions in Object-Oriented Databases. Technical Report UTC/GI/DI/N 83, Universite de Technologie de Compiegne, 1991.

[Bee89] C. Beeri. Formal Models for Object-oriented Databases. In *Proc. 1st International Conference on Deductive and Object-oriented Databases*, pages 370–395, 1989.

[Bla91] H. Blanken. Implementing Version Support for Complex Objects. *Data and Knowledge Engineering*, pages 1–25, 1991.

[BM91] C. Beeri and T. Milo. A Model for Active Object-oriented Databases. In *Proceedings 17th VLDB*, pages 337–349, 1991.

[BS81] F. Bancilhon and N. Spyratos. Update Semantics in Relational Views. *ACM TODS*, pages 293–305, December 1981.

[CJ90] W. Cellary and G. Jomier. Consistency of Versions in Object-Oriented Databases. In *Proc. 16th VLDB*, pages 432–441, 1990.

[CJK91] W. Cellary, G. Jomier, and T. Koszlajda. Formal model of an object-oriented database with versioned objects and schema. Work in Progress; unpublished Draft, 1991.

[CVJ91] W. Cellary, G. Vossen, and G. Jomier. Multiversion Object Constellations for CAD Databases. Technical Report 9105, Justus-Liebig Universitat Giessen, 1991.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. ACM SIGMOD*, pages 36–58, 1990.

[EL90] J. L. Encarnacao and P. C. Lockemann. *Engineering Databases - Connectiong Islands of Automation Through Databases*. Springer-Verlag, 1990.

[GJ91] N. Gehani and V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings 17th VLDB*, pages 327–336, 1991.

[JMSS90] M. Jarke, S. Mazumdar, E. Simon, and D. Stemple. Assuring Database Integrity. *J. Database Admin.*, 1(1):391–400, 1990.

[Kat90] R. H. Katz. Toward a Unified Framework for Version Modelling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, 1990.

[KDM88] A. Kotz, K. Dittrich, and J. Mulle. Supporting Semantic Rules by a Generalized Event/trigger Mechanism. In *Proc. 1st EDBT*, pages 76–91, 1988.

[KS92] W. Kafer and H. Schoning. Mapping a Version Model to a Complex-Object Data Model . In *Proc IEEE Data Engineering Conference*, pages 348–357, 1992.

[KSW86] P. Klahold, G. Schlageter, and W. Wilkes. A General Model for Version Management in Databases. In *Proc XII VLDB*, pages 319–327, 1986.

[MP91] C. Medeiros and P. Pfeffer. Object Integrity Using Rules. In *Proceedings European Conference on Object-Oriented Programming*, pages 219–230, 1991.

[Sci91] E. Sciore. Multidimensional Version for Object-Oriented Databases. In *Proceedings VLDB*, pages 355–370, 1991.

[Sno90] Richard Snodgrass. Temporal Databases Status and Research Directions. *SIGMOD Record*, 19(4):83–89, December 90.

[TG92] V. Tsotras and B. Gopinath. Optimal Versioning of Objects . In *Proc IEEE Data Engineering Conference*, pages 358–365, 1992.

Relatórios Técnicos

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in d-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An (l, u)-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

06/92 **Implementing Integrity Control in Active Databases,** *C. B. Medeiros, M. J. Andrade*

07/92 **New Experimental Results For Bipartite Matching,** *J. C. Setubal*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

reltec@dcc.unicamp.br