Versions, Configurations, and Constraints in CEDB[1]

H. Craig Howard[2,3]
Arthur M. Keller[4]
Ashish Gupta[4]
Karthik Krishnamurthy[2]
Kincho H. Law[2]
Paul M. Teicholz[2,5]
Sanjai Tiwari[2]
Jeffrey D. Ullman[4]

Abstract

The architecture-engineering-construction (AEC) industry is highly fragmented, both vertically (between project phases, e.g., planning, design, and construction) and horizontally (between specialists for the various disciplines at a given project phase, e.g., design). We need software that detects, analyzes, and manages changes efficiently during concurrent distributed design processes. In the CEDB (Collaborative Environment for the Design of Buildings) project, we have developed a model of a combination of versions, configurations, and constraints. Versions are organized into hierarchies of alternatives within a single discipline (e.g., architecture, structural engineering). A configuration is a set of versions, one from each of a number of disciplines, combined with a set of cross-disciplinary constraints to check for violations. Our objective in this integrated model of versions, configurations, and constraints is to assist designers by informing them of the changes by others that affect them and their changes that affect others, in particular, the changes that result in constraint violations. To accomplish this objective, we aim to find those violations as efficiently as possible. We have substantially implemented our model using multiple ORACLE databases.

---

1 This work is part of the CEDB project (Collaborative Environment for the Design of Buildings, or Civil Engineering DataBase), Jeffrey D. Ullman, Principal Investigator. This effort is funded in part by National Science Foundation grant IRI-91-16646.

2 Department of Civil Engineering, Stanford University, Stanford, CA 94305-4020

3 Phone: 415-723-5678; e-mail: hch@cive.stanford.edu

4 Department of Computer Science, Stanford University, Stanford, CA 94305-2140

5 Center for Integrated Facility Engineering, Stanford University, Stanford, CA 94305-4020

1. Introduction

The U.S. architecture-engineering-construction (AEC) industry is highly fragmented compared with many of its Asian and European competitors [Howard 89a]. This fragmentation exists both within individual phases of the construction process (e.g., the design phase), and across project phases, from planning through design and construction and into facility maintenance and operation. The problems arising from fragmentation affect productivity and competitiveness throughout the AEC industry. Our goal in this work is to address this fragmentation through change management tools that facilitate and enhance collaboration among multiple designers and contractors.

To provide change management in this environment, we propose a combination of versions, configurations, and constraints. A *version* represents a design checkpoint in a single discipline (e.g., architecture, structural engineering). Versions may be grouped hierarchically within a discipline, with later versions represented as modifications to earlier versions and parallel alternatives represented as branching versions. *Constraints* are used to manage interactions across the discipline-specific versions by specifying inconsistent design states—combinations that aren't allowable. A *configuration* represents an "integrated" design, with one version from each of a number of distinct disciplines combined with a set of constraints to be verified. A configuration can be generated for major design review or in response to a "what-if" test by a single designer who wants to compare his/her latest version with existing versions from other disciplines. A configuration results in a list of violations that trigger notifications to the designers involved in the constraints. By providing operators to compare the versions within new configuration to the versions in a previously checked configuration, we can perform incremental constraint checking, thus limiting the volume of changes that need to be tested in each design iteration. Our objective in this integrated model of versions, configurations, and constraints is to notify the appropriate designers of constraint violations and to find those violations as efficiently as possible.

This paper describes the CEDB (Collaborative Environment for the Design of Buildings or Civil Engineering DataBase) project at Stanford University, an interdisciplinary effort between the departments of computer science and civil engineering, that addresses change management for concurrent design. Section 2 discusses some of the related work in the areas of versions, configurations, and constraints. Then Section 3 provides an overview of the AEC domain, including a pair of sample databases and a sample constraint based on a real example. Sections 4 and 5 describe the model in detail and demonstrate the processing of changes using the example defined in Section 3. Finally, we conclude by summarizing the salient points of the model.

2. Related Work

For versions and configurations, there is a useful survey in the area [Katz 90]. Katz considers the following issues in version and configuration management: (i) organizing the version set, (ii) static and dynamic binding mechanisms, (iii) hierarchical compositions, (iv) version grouping mechanisms, (v) change notification and propagation, and (vi) object sharing mechanisms. The following issues addressed by our work are surveyed in detail by Katz:

- **Versions of an individual entity:** Katz et al. have formalized the version derivation history as a hierarchy [Katz 86]. More general structures as a rooted DAG have been proposed in [Klahold 86, Ecklund 87]. The versions are connected by *derived-from* links.

- **Versions of an assembly of entities:** Previous research efforts have defined *configurations* as the version of a composite entity in terms of the versions of its components [Katz 87, Ketabchi 87, Landis 86, Lorie 83].

- **Inheritance among versions:** *Type-version* inheritance [Batory 85], as well as *instance-instance* inheritance schemes along *descendant-of*, *equivalent-to*, and *component-of* relationships [Katz 89] have been proposed.

- **Implementation Schemes:** Implementation of versions in terms of *deltas* to support the incremental addition of data to a version has primarily been studied for the software engineering environment [Rochkind 75, Leblang 84]. There has been relatively little effort on developing versioning systems that support the evolutionary nature of the design process, i.e., the incremental addition of data to an individual version.

The subject of constraint management has been a very active area in engineering automation, and we will concentrate here on a few of the civil engineering research projects that have directly influenced our effort. Holtz [Holtz 82] described symbolic algebra and dependency driven expressions to manipulate constraints for consistency management in design applications. Rasdorf and Fenves [Rasdorf 86] emphasized the importance of automated representation and processing of design constraints in relational databases. They proposed a mechanism of augmented relations, where additional attributes were appended to the database relation schemes, Their work assumes a centralized database. The DICE project [Sriram 92] addresses the issues of coordination and communication through a centralized blackboard and a global project database. DICE includes conflict handling in shared transactions, where the participants are notified of each update made within the scope of a transaction. The DICE project studies transaction management and concurrency control for collaborative engineering environments, while we focus on efficiency aspects of real-time constraint management and collaboration.

Most of the work done in integrity constraint management concentrates on centralized databases. However, the issues involved in checking constraints that span multiple databases have not received much attention. Yet distributed constraints are essential in a design environment composed of independent disciplines that need to coordinate.

The architecture proposed in this paper can use most of the existing approaches to constraint management at the local sites. If an underlying system supports local constraint management, then we incorporate that capability into the global validation process. Our constraint management system can be built on top of the existing database systems. Ceri and Widom [Ceri 90] describe a production rule system that allows declaration of rules that are triggered on events and their corresponding actions executed if some conditions are met. Such a system can provide monitoring at any of the underlying sites. In fact, part of our implementation uses their system as the backend. [Qian 88] describes techniques for distributing global constraints among sites in a way that reduces communication at run time. These techniques can be used to preprocess constraints in our architecture. Similarly, efficient constraint checking techniques discussed in [Nicolas 82, Bry 92] can be used by both local constraint managers and the global constraint manager. Distributed constraint checking can also be made more efficient by using the demarcation protocol [Barbara 92] or by generating queries that are sufficient to allow us to infer that a constraint has not been violated [Blakeley 89, Gupta 93b, Levy 93].

3. The Problem Domain

To provide the reader with a more tangible sense of the data management issues, Figure 1 illustrates several of the key stages involved in a building project. The figure emphasizes the differing views that the various project participants have about the data describing the process. The following discussion elaborates on the management of data at each stage of the process:

- The *architect* may start with grand ideas for the structure, but must shape these into a more practical design based upon the constraints (financial, schedule, technical, operational) generated by the owner, engineers, users and others in the process. The architect is normally the overall coordinator and leader in this multidisciplinary process, and increasingly is likely to use advanced CADD software. The architect also generates many, if not most, of the constraints and criteria that govern others in the process, but currently communicates with them mainly through traditional paper documents and meetings.

- The *structural engineer* takes the architect's ideas and designs the skeleton that allows the building to resist the loads that derive from its function (e.g., furniture and people) as well as the loads caused by the environment (e.g., wind and earthquakes). As with the architect, the operations of the structural engineer may be highly computerized, but the coordination of changes is still currently a manual process.

- The *contractor* takes the contract plans and specifications that result from all of these prior deliberations, and adds his or her knowledge of costs, schedules, methods and materials. The result is an estimate and resource-based schedule for completing the project most efficiently. The general contractor normally retains specialty subcontractors who, in turn, may further subcontract specific tasks. There is little use of electronic data communication across these boundaries, and coordination of changes is a time-consuming and error-prone process—frequently involving the use of a jackhammer.

- The *owner* inherits the resulting performance of the structure (which hopefully stands up better than shown in the Figure 1). The owner's facility management must start with an accurate record of what was designed and built. That record must then be maintained over the lifetime of the facility. Many facilities undergo one or more modifications during their lifetime, particularly manufacturing or chemical process facilities such as computer chip fabrication plants, refineries, and space launch facilities. Even more mundane facilities may require renovation for new uses, retrofitting for seismic upgrade, etc.

The intent of this discussion is to highlight the differing views of each of the project participants, to emphasize the current lack of electronic data communications among the participants, and to underscore the potential for automated change management. The following section summarizes the requirements for change management in the AEC industry. Section 3.2 then introduces an example from the AEC domain that will be used throughout the rest of the paper.

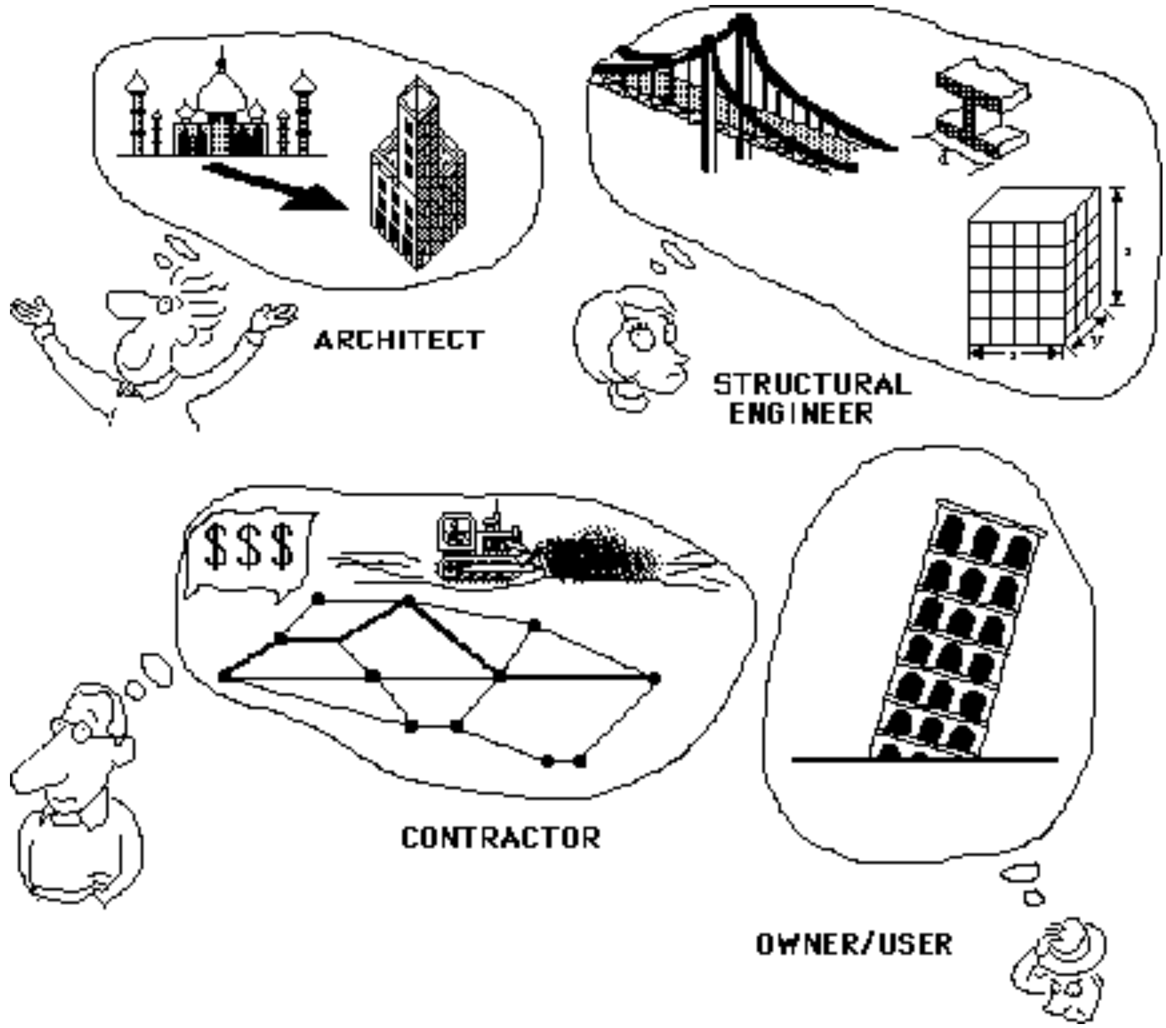


Figure 1: Differing Views of the Project Data

3.1. Requirements

To allow a project team consisting of the owner, architect, engineers, and contractors to work together in an efficient and cost effective manner requires changes in both technology and business relationships. In this paper we will concentrate on the technological requirements to support the concurrent design efforts of a distributed design team, which include the following:

- Identify when constraints have been violated by any of the team members and notify the appropriate team members in a meaningful way (design solutions are the responsibility of the team members);
- Allow periodic checkpoints of the design so that it is possible to control the design as it develops over time;
- Provide a mechanism for efficiently identifying the changes that have occurred between any checkpointed versions of the design;
- Provide security of the design environment for each of the team members (i.e., designers may change elements only within their own disciplines); and
- Allow negotiation over design alternatives among the distributed team members.

3.2. Introductory Example

To provide context for our approach to change management, we will use the following running example based on Tiwari's [Tiwari 93a] exploration of automated constraint checking for space-launch facility and power plant projects. Consider a space launch complex under construction. There will be bridges connecting buildings of the complex. There will also be trucks (tankers, vans, etc.) that will go underneath these bridges (see Figure 2). The architect maintains the database of bridges, including their clearances above ground. As the design evolves, new bridges are added, and bridges already in the design may be raised or lowered. The evolution of the architect's design is represented by a series of versions describing successive refinements of the design or design alternatives. The owner already has trucks to be used in this complex. New trucks will be purchased and old ones will be sold. The owner maintains a database of trucks. The truck database is also versioned, as it evolves along with the owner's planning process. A sample set of data and changes to the architect's and owner's databases is shown in Table 1.

There are configurations that consist of one version of the architect's database and one version of the owner's database, along with the constraints that this configuration must satisfy. In this example, the constraint is "all trucks must fit under all bridges." We assume that the architect's database and the owner's database are stored on separate computers connected by a wide area network.

We can make several observations. First, we can create a configuration incrementally. That is, a configuration can be based on a prior configuration with certain changes made to the constituent versions. Second, when configurations are made incrementally, we can check the constraints incrementally. That is, when the change that resulted in the new configuration is to add a new bridge, we need only check that this new bridge does not violate the constraints, perhaps by checking its clearance against the heights of all the trucks. Third, we may be able to check a constraint locally, i.e., in the database that was changed. For example, if the new bridge has a higher clearance than some existing bridge, and the existing bridge did not violate a constraint (i.e., it had enough clearance for all of the trucks), then we can deduce that the new bridge has sufficient clearance without checking the height of any truck. We can further complicate this example by defining separate zones in which bridges may be placed and through which certain trucks must traverse; in our sample data, we'll stick with a single zone to simplify the presentation, but we will discuss the constraint processing for the more complicated case. This example and these observations will be explored in further detail in the subsequent sections of this paper.

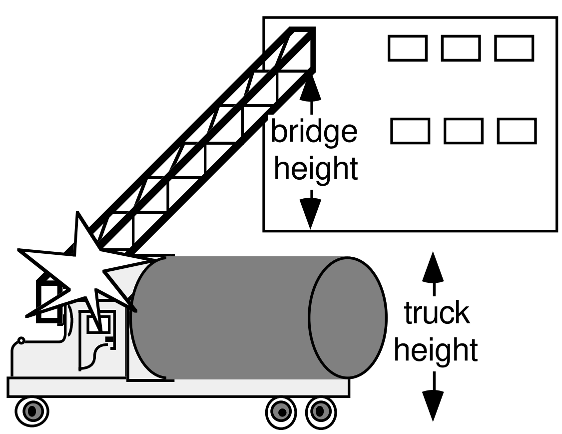


Figure 2: Trucks and Bridges Example

Architect's Database				Owner's Database		
Bridges relation				Trucks relation		
bridgeID	clearance (meters)	zone		truckID	height (meters)	zone
B1	4.5	1		T2	4.0	1
B3	5.0	1				
B4	5.0	1	change 1: insertion			
B5	4.0	1	change 2: insertion			
			change 3: insertion	T3	4.5	1
B5	4.5	1	change 4: modification			

Table 1: Sample Data and Changes for Bridge and Truck

4. Combined Model for Versions, Configurations, and Constraints

Our model involves a distributed set of databases, usually one for each discipline. The specialists in a discipline have read and write access in their own databases and some level of read (but not write) access to other databases. Each discipline's database has its own version hierarchy. There is also a version hierarchy of constraints, and a constraint can apply to one of more versions. A *configuration* consists of one version from each design database version hierarchy, plus a constraint version, as suggested by Figure 3. Each configuration can be based on a prior configuration. Our approach is to check constraints incrementally, so that constraints already used to validate the prior configuration need only be checked against the changes between new and old versions for each discipline whose version changed in going from the old configuration to the new. Furthermore, the changes can often be checked within one design database, even when the constraint affects more than one design database. New constraints must of course be checked against the entire new configuration. The remainder of this section will describe the model for distributed versions, configurations, and constraint checking.

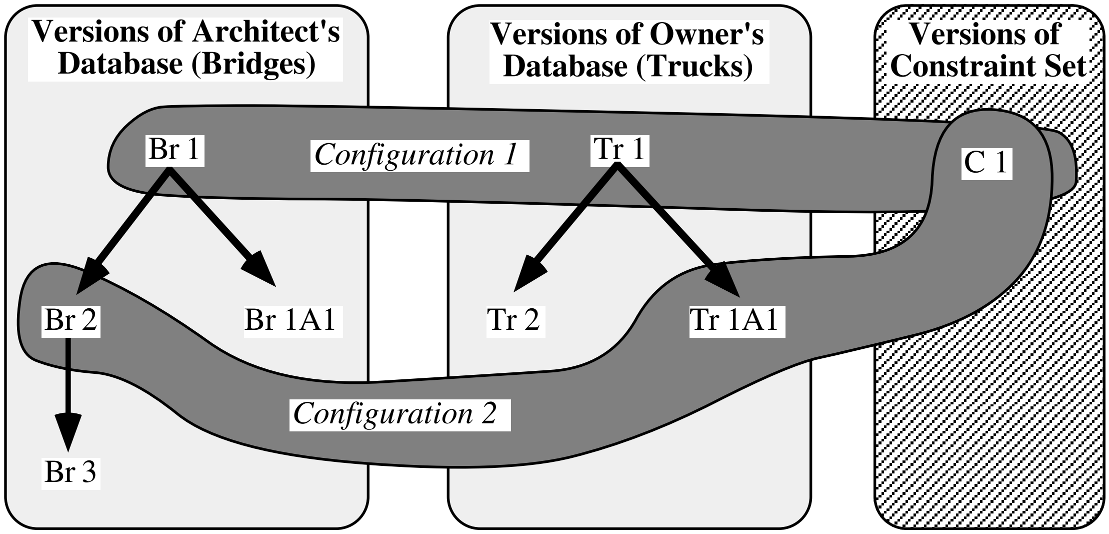


Figure 3: Overview of Versions and Configurations for Sample Data

4.1. Versions

During the design process, application transactions incrementally modify a design database with design changes (*deltas*) that represent operations (add, delete, or modify) on the database contents. In our model, a version represents a specific design alternative in a particular discipline's database [Krishnamurthy 93, 94a,b]. Versions may be created, suspended, activated, declared, derived, and removed, as shown in Figure 4. The first step is to *create* an initial *active* version, which may be modified by application transactions to form a starting design. That version may be *suspended* so that some other version may be the focus of new design transactions, or it may be *declared* so it can be accessed but not altered. The *derive* operator is used to define a new version that is an extension of some previously declared version. Since the derive operator can be applied to any previous version, we can easily define hierarchies of branching alternative versions. In Figure 4, version state operators are represented by directed arcs, while the *derive* operator that links two versions is represented by a dashed arc.

Figure 5 shows the sample version hierarchies using our example data. In the hierarchy for the architect's database, the original set of data constitutes version Br1.[1] Once that version is *declared*, a new version Br2 can be derived. Data operations add to Br2 the new tuples corresponding to changes 1 and 2. In parallel, version 1A1 explored the addition of bridges B2 and B4, but was *suspended*. Finally, version 3 incorporates the height modification for bridge B5 resulting from change 4. Within the version hierarchy, a bridge entity is uniquely identified by bridgeID and version number; for example, <B4, Br2> and <B4, Br1A1> have different heights.

1 The numbering scheme for new versions is described in [Keller 94].

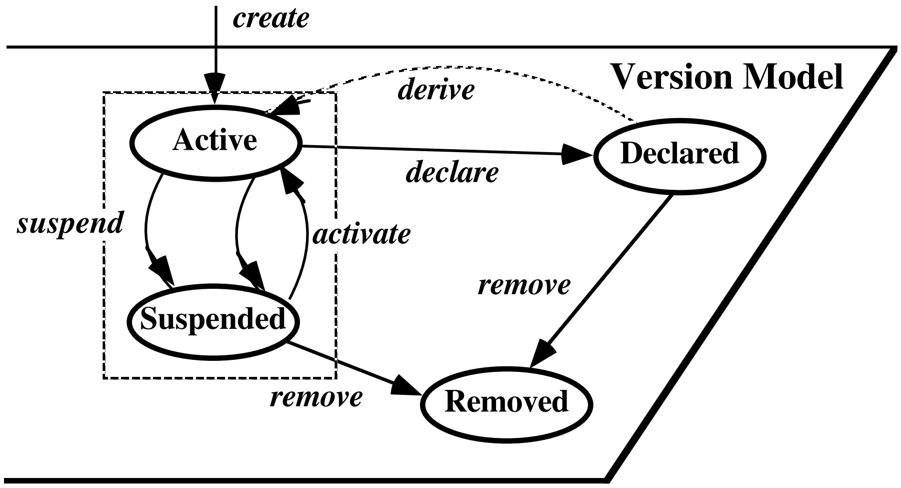


Figure 4: Model of Versions to Support Design Applications

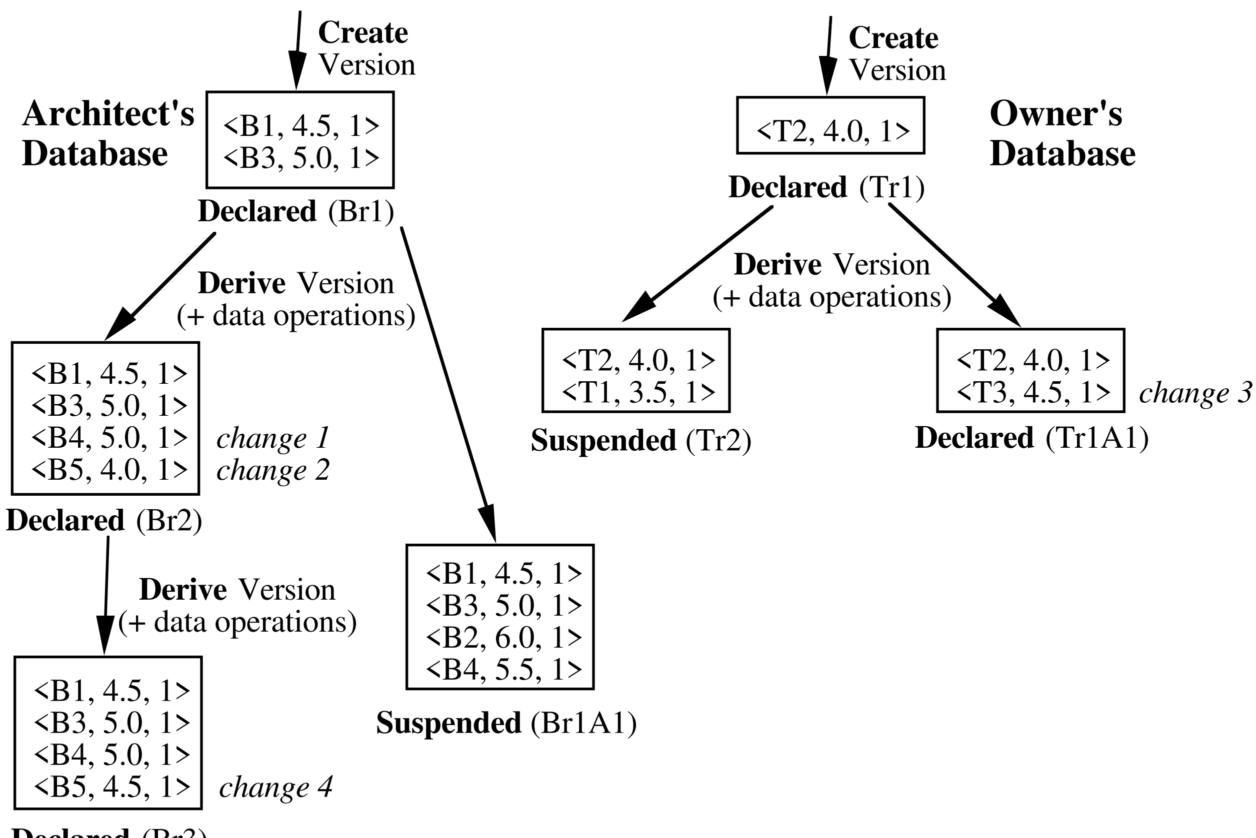


Figure 5: Sample Version Hierarchies

bridgeID	clearance	zone	operation description
B4	5.0	1	insert
B5	4.5	1	insert

Table 2: Deltas Computed Between Version Br1 and Br3

There are two main approaches to capturing the changes from the design process. One approach is to capture changes as they are made to a design. This approach requires that version management be integrated into the design process and be supported by the design software. Unfortunately, this requirement severely limits the choice of design software. The approach we choose is to use a check-out/check-in sequence for each design process. A designer will check-out a design, make changes to it using the user's preferred software, and then check-in the modified design. It is then the job of version management to compare the version checked out with the version checked in subsequently to determine what changes have been made. Our version management model actually supports both approaches, as the distinction is limited to the nature of updates to an active version.

Additional operators are provided to query the contents of any existing version and to determine the differences between any two hierarchically linked versions. In particular, the *compute deltas* operator condenses the differences between two versions into a minimal set of data operations (deltas) that, when executed on the ancestor version, produce the child version. As an example, Table 2 shows the deltas (changes) computed when comparing Br1 with Br3.

Another group of version operators is required to handle version status and version access privileges. However, we must first explore the configurations in order to understand why those status and access operators are needed.

4.2. Configurations

A configuration is a framework to integrate designs from the different disciplines to describe an overall project design. Versions represent individual designs, and constraints specify the restrictions among them. Thus, a configuration is formally specified as a set of versions (at most one from each discipline) and a set of applicable constraints [Krishnamurthy 93, 94a,b]. A configuration may be defined formally to check the integrated project data at major project milestones or informally to test a specific designer’s alternatives against other disciplines. In either case, the configuration definition will produce the set of constraint violations present in the integrated data set.

The operations for configurations are summarized in Figure 6. The initial configuration is created using the *define* operator. Later configurations may be incrementally *generated* from earlier configurations or defined afresh. To generate a new incremental configuration, each database version in the new configuration must be a descendant of the corresponding discipline-specific version from the previous configuration. Generated configurations can streamline violation detection by focusing constraint checking on the changes between the previously checked configuration and the new configuration. Once a configuration has been generated, the result is no different from a configuration created using the define operator. Figure 3 illustrated two sample configurations defined using our sample databases—configuration 1 was initially defined and then configuration 2 was generated from configuration 1.

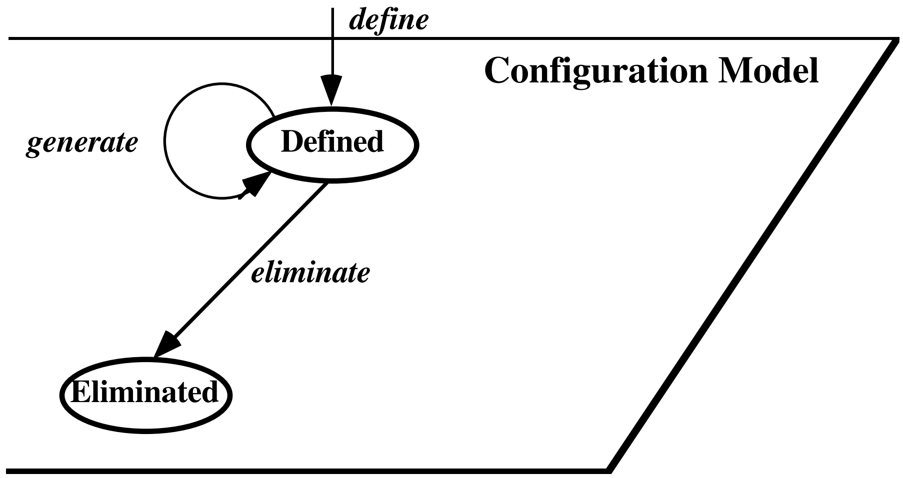


Figure 6: Model of Configurations

A configuration is a static structure and hence cannot be modified, but we do define operators to access the contents of configurations. In particular, the *characterize-config-deltas* operator enumerates the changes between two configuration definitions, where each version in one of the configurations is an ancestor of the version from the same discipline in other configuration. The changes are expressed in terms of the deltas between the corresponding component versions and differences between the associated sets of constraints. The resulting change information is input for the constraint manager presented later in this section and demonstrated in Section 5.1.

4.3. Version Model to Support Configurations

A version must be guaranteed to exist while it is included in a configuration definition. Additionally, the designer must be able to access the contents of a version from another discipline to include it in a configuration definition. To satisfy these conditions, two additional properties are specified for versions: *version status* and *version access privileges*. The former refers to the protection status of a version, and the latter to the accessibility of its contents to users in other disciplines. To incorporate those properties, we add two states to the model shown in Figure 4 to form the final version model shown in Figure 7. A declared version can be *frozen* to indicate that it cannot be removed. A frozen version can be *published* to make it accessible to users in other disciplines. When a designer defines or generates a configuration, any versions from other disciplines must be published before they can be included in the configuration; versions from the designer's own version need only be frozen. Before a version can be *suppressed* or *thawed*, any configurations in which it participated must be eliminated.

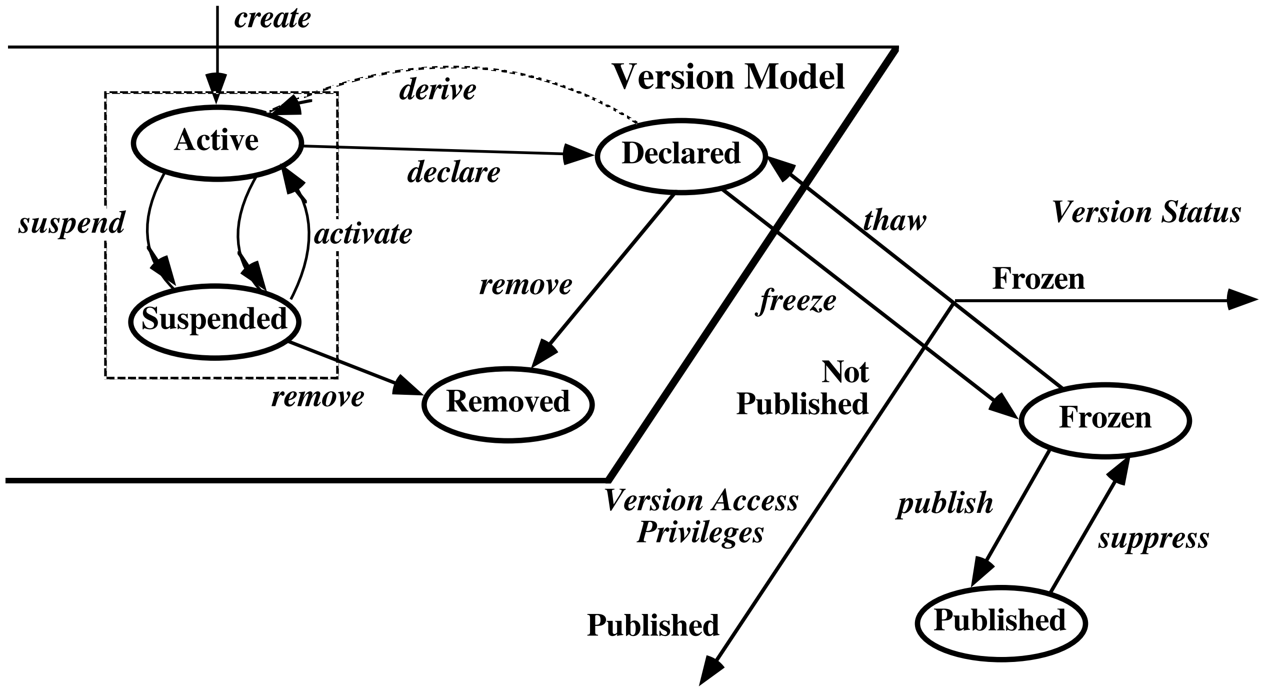


Figure 7: Model of Versions to support Configurations

4.4. Constraints

The project participants must check the consistency of their design periodically with respect to a set of configuration constraints. These constraints can arise from high-level functional requirements, such as those specified by the client, or from the interface requirements of different engineering systems and components. For example, in AEC projects, constraints deal with objects across different disciplines such as architectural, structural engineering, mechanical, electrical and plumbing [Tiwari 93]. Similar to the design data, project constraints also evolve as the design progresses from the conceptual phase to detailed design, and they can change from one project to another. Therefore, the configuration model must include versions of the constraint set, as noted in Section 4.2.

We use constraints as a means to model the design dependencies among objects. Often constraints cross disciplines, although "local" constraints involving a single discipline are also important. The approach of making dependencies orthogonal to the representation of design data has two advantages:

- **Uniform Representation:** Interdisciplinary design constraints are uniformly expressed on design (database) states and specify “what” (declarative specification) to enforce rather than “how” (procedural specification). A high level representation of constraints alleviates the problems of obscure semantics for *ad-hoc* user-programmed constraints.

- **Optimization**: A declarative representation of constraints affords many opportunities for performance improvements and optimizations [Ullman 89]. A constraint language can assist in real-time management of design constraints by enabling the mapping from “what” a constraint means to “how” to enforce it at the execution level.

Constraints on databases have an associated state. At any given stage of the design process, a constraint can be either in a satisfied or a violated state. The state associated with a constraint may change due to design changes (insert, delete, and update) made to an object or set of objects. In our model, a constraint specification identifies inconsistent design states, i.e., the condition that becomes true upon a constraint violation. In a collaborative design scenario, specifying inconsistent states is more natural and efficient than enumerating all possible consistent states. We extend the language in [Ceri 90] to specify the participant who creates the constraint, the participant responsible for violations, the content of the notifications, and the participants to be notified. The use of a high-level constraint language allows many compile-time optimizations that make use if the information about the distribution of design data between different databases, as we will see in Section 5.

Consider our earlier constraint on a configuration where information about the bridges is stored in the architect's database and information about the trucks is stored in a owner's database. Constraint 1 requires that every truck should be able to pass under all the bridges in its service zone. The constraint is violated if the height of some truck is greater than the clearance of some bridge in the service zone of that truck. We use an extension of SQL to express the constraint as follows (the "::" symbol is used to associate the database name, e.g., the database of the "Owner," with the relation name) [Ceri 90, Gupta 93a, Tiwari 94]:

```
Owner::Trucks.Height > any
    ( select  Bridges.Clearance
      from    Architect::Bridges
      where   Bridges.Zone = Trucks.Zone)
actions:
      Notify(Architect, Owner);                                    (1)
```

4.5. Architecture of Constraint Management System

Many commercial databases incorporate attribute-level constraints on database relations, for example, that the bridge clearance should be at least 3 meters. At times, this facility may suffice within a design domain, since domain-specific constraints are normally checked within the application. However, managing constraints across different databases introduces many other variables into the picture. To elaborate on the general requirements presented in Section 3.2, we identified four objectives for constraint management in a collaborative AEC design framework:

(1) the system should handle constraints involving different databases, which may be logically and physically distributed;
(2) the system should be able to store and check most of the complex constraints expressed in a language with at least the power of relational algebra;
(3) the system should compile these constraints in order to maintain them efficiently during the run-time of applications; and
(4) the system should support local autonomy and emphasize distributed processing for enhanced performance.

Figure 8 shows such a framework with autonomous architect, structural and owner databases [Tiwari 94]. A global constraint manager (GCM) is responsible for maintaining a set of inter-domain constraints, and it uses the facilities provided by the local constraint managers (LCMs) and monitors. A description of the components of the constraint management system follows:

- **Constraint:** A constraint expresses an invalid design state over a set of autonomous design databases. The inter-domain constraints can involve multiple design attributes of objects residing in different databases. In our approach, a high-level constraint language is used to specify constraints to the global constraint manager. The constraints are declarative in that a user specifies "what" is required to be enforced rather than how to enforce a constraint. Frequently, the constraints are not known a priori, i.e., at the time of database design. The constraint language provides *data independence* in this sense, constraints need not be put into the system with the data.

- **Application:** An application program needs input data for domain-specific reasoning and analysis. The output of an application (e.g., a structural analysis or a material cost estimate) usually involves refinement or addition of design attributes to the database objects. Applications interact with the DBMS through transactions managed by the Version Manager as described in Section 4.1.

- **Global Constraint Manager:** A global constraint manager is entrusted with the task of decomposing and distributing constraints. The global constraint manager (GCM) is distinguished from local constraint managers by having a catalogue of all the design information at various sites (a global data dictionary). The GCM also has a repository of all the design constraints and provides facilities for updates or queries on constraints themselves. Thus, constraints are first class database objects. The constraint compiler resides within the GCM. It extracts the relationships between constraints and database objects, generating local constraint fragments for each database that contains objects referenced in the constraints, monitors to watch for potentially invalidating operations (see below), and a global constraint fragment to check the constraint with data obtained from the distributed database. (The constraint compilation process is described in Section 5.1.)

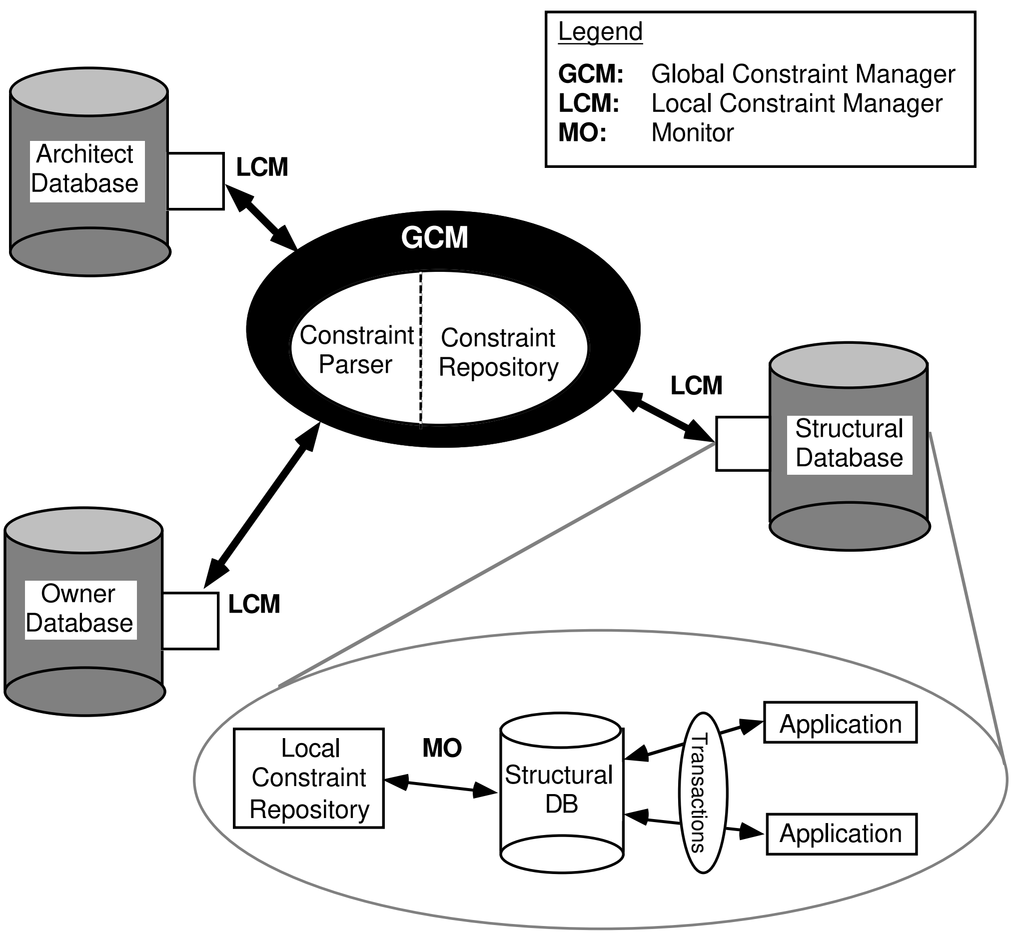


Figure 8: Architecture of a Distributed Constraint Management System

- **Local Constraint Manager:** Local constraint managers (LCM) are stripped-down versions of their global counterpart and manage portions of constraint specifications relevant to their site. LCMs communicate directly with the local database and query the data in order to check constraint violations. They can also send the relevant data to the GCM if the need arises. The primary function of an LCM is to maximize local constraint validation, communicating the minimal required data to the GCM if local validation is not sufficient. LCMs also make use of integrity checking facilities provided by local DBMS .

- **Monitors:** As the name suggests, monitors provide a monitoring facility for individual design objects in a database. There is one monitor per design object. Monitors interact closely with the local DBMS and are constantly on the lookout for updates to the design objects within a transaction that may lead to a constraint violation. The relevant update information is communicated to the LCM if it is a potentially invalid update.

4.6. Notifications

In general there can be a large number of constraints and associated violations in a project. The participants may not be aware of all the constraints defined on their databases, since they may be specified by other participants or disciplines. Therefore, we need a constraint classification scheme by which participants can be efficiently notified of constraint violations. For the purpose of classifying constraints in distributed configurations, we assign additional attributes to the database objects, constraints and violations that are the components of a configuration. Objects are assigned an *owner* attribute to identify who can make changes to the object; constraints are assigned a *source* attribute to identify who created a constraint, and violations are assigned a *responsibility* attribute to identify who is responsible for correcting that violation. Different possible combinations of these attributes for configuration components are used to classify constraints as shown in Table 3.

We make the local-global distinction since global constraints (involving multiple remote databases) are usually more time consuming to check than the local ones that involve no remote data access. In addition, participants are usually more familiar with the constraints *internal* to the discipline than the *external* constraints imposed by participants belonging to different disciplines. Designers may prefer to focus on the constraints that originate from within their own discipline.

Constraint Example	**Constraint Source**	**Violation Respon-sibility**	**Object Owner(s)**	**Constraint Type**
Check that all door frames for the personnel rooms in the architectural database are made of fiberglass material.	**X** *Architect*	**X** *Architect*	**X** *Architect*	**Local-Internal**
Check that all the equipment in the mechanical database is connected to the electrical grounds in the electrical database.	**X** *Mechanical*	**X** *Mechanical*	**X & Y** *Mechanical & Electrical*	**Global-Internal**
Check that there are no conduits below 7'2" in the mechanical database.	**X** *Safety*	**Y** *Mechanical*	**Y** *Mechanical*	**Local-External**
Check that the capacity of each crane in the contractor's database is greater than the weight of the columns in the structural database that it has to lift.	**X** *Project Manager*	**Y** *Structural*	**X & Y** *Structural & Contractor*	**Global-External**

Table 3: Constraint Classification to Handle Notifications

5. Efficient Constraint Management

When a new configuration is generated from a previous configuration, the constraints need to be reevaluated and a new set of violations needs to be computed. Consequently, constraint checking can be very expensive when the configurations are very large; i.e., the individual databases could be large and there could be many constraints. The expense of checking constraints can be reduced if we consider only the *changes* made to the configuration and do not evaluate all the constraints on the entire databases belonging to a new configuration.

To provide efficient constraint checking, various optimization strategies can be used at the time that constraints are specified (*compile-time*) and the time that constraints are checked (*run-time*). The underlying databases are used to store the optimization information derived at constraint compile time, in *constraint repositories*. At run-time, the constraint managers access the derived information in repositories using the local database management system. The following sections describe the compilation and run-time processing as well as the status of our implementation.

5.1. Constraint Compilation and Fragmentation

Even though constraint checking is performed when the configuration changes, the constraints are preprocessed (or compiled) in order to make the checking process efficient. The compilation phase extracts information needed for constraint checking. Constraint compilation produces procedural specifications and run-time optimizations from a high-level constraint specification. The compilation process fragments a global constraint to produce database-specific local components. Fragmentation reduces the amount of information that the local databases need to send over the network for global constraint checking. The information produced by the compilation phase is shared between the GCM and the site-specific LCMs. The GCM and LCMs store this derived information in repositories that can be queried and used efficiently at run-time when constraint checking takes place.

In the next section, we illustrate the information derived at compile-time and describe how this information is used to check constraints. In [Tiwari 94], we describe the compilation process in detail.

5.2. Run-time Constraint Processing

Run-time optimizations can be used to avoid unnecessary checks and data transfer. Figure 9 graphically shows our objective to reduce the volume of data that the human designers need to process in change management. Rather than having to sift through the complete mass of data defined in a new configuration, the human designers can concentrate on those changes that violate constraints. The figure also summarizes the steps involved in the runtime constraint management that are described in the following sections.

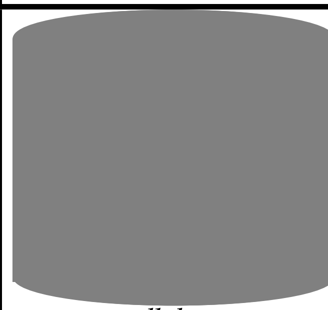
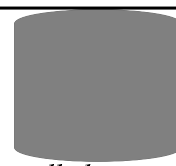
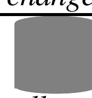
Action	Size of Data Set
Configuration Manager — Declare a new configuration	 *all data*
Configuration Manager — Identifies changes (deltas)	 *all changes*
Change Monitors — Identify potentially violating changes	 *potential violating changes*
Local Constraint Managers — Eliminate locally verifiable changes	 *nonlocally verifiable changes*
Global Constraint Manager — Identify global constraint violations and generate notifications	 *violations (notifications)*

Figure 9: Run-Time Processing

5.2.1. Changes

Design changes originate with the design transactions applied to specific versions. However, interdisciplinary constraints are not checked until a configuration is defined or generated. When an incremental configuration is generated, the configuration manager is responsible for determining the changes (deltas) between the previous configuration and the new configurations. For our run-time example, consider configurations 1 and 2 as originally defined in Figure 3 and summarized in Table 4. Configuration 1 satisfies the current constraint (constraint 1) and, therefore, has no violations. When configuration 2 is defined as a descendant of configuration 1, the first step in the constraint checking is to determine the changes from configuration 1 to configuration 2 by comparing each version with its ancestor using the *compute deltas* operator for versions as described in Section 4.1. The result is shown in Table 5. By looking at only changes, we've cut the volume of data to check in half in our tiny example; in practice, the number of changes between configurations would typically be much smaller than the overall quantity of data.

configuration	parent configuration	Architect's DB version	Owner's DB version	constraint set	violations
configuration 1		Br1	Tr1	constraint 1	none
configuration 2	configuration 1	Br2	Tr1A1	constraint 1	?
configuration 3	configuration 2	Br3	Tr1A1	constraint 1	?

Table 4: Sample Configuration Definitions

version operator	bridgeID	clearance	zone	data operation
compute-deltas (Br1, Br2)	B4	5.0	1	insert
	B5	4.0	1	insert
version operator	**truckID**	**height**	**zone**	**data operation**
compute-deltas (Tr1, Tr1A1)	T2	4.5	1	insert

Table 5: Changes (Deltas) Computed for Configurations 1 and 2

5.2.2. Potentially Violating Changes

Not all database changes have the potential to violate constraints. For instance, constraint 1 may be violated if the owner introduces a new truck, or if the architect updates the clearance of a bridge in any zone. Conversely, the deletion of a bridge or truck cannot possibly violate the constraint. In general, for any constraint expressed in our language, it is possible to identify the database operations that could potentially violate that constraint. These potentially violating operations are derived at compile time. For constraint 1, the potentially violating operations are given in Table 6. Each potentially violating operation is specific to either the architect's or the owner's database. The architect need not be aware of the owner's potentially violating operations and vice-versa. Therefore, the set of potentially violating operations can be fragmented and distributed between the two databases to be handled by the monitors within the local constraint managers. In our comparison of configurations 1 and 2, all of the changes presented in Table 5 correspond to potentially violating changes. The first two are local to the architect's database; the third is local to the owner's.

Architect's Database	`updated Architect::Bridges.Clearance` `inserted Architect::Bridges`

Owner's Database	`updated Owner::Trucks.Height`
	`updated Owner::Trucks.Zone`
	`inserted Owner::Trucks`

Table 6 : Potentially Violating Operations

5.2.3. Locally Verifiable Changes

Even though a constraint refers to multiple databases, the constraint can often be checked using the data from only one of the participating databases. For instance, let a new bridge *B* be placed in zone *Z*. If constraint 1 was previously satisfied and if there is another bridge that is lower *than* bridge *B* in zone *Z*, then we can conclude that bridge *B* must be higher than every truck that enters zone *Z*, and therefore, the constraint is still satisfied. Thus, constraint 1 could be checked without accessing the owner's database. We refer to these conditions as *local tests*. Different local tests are derived, at compile time, for different potentially violating updates. For the case of insertions into relation Architect::Bridge, the local test is:

```
Select   *
From     Architect::Bridges
Where    Bridges.Zone = new_bridges.Zone
   and   Bridges.Clearance <= new_bridges.Clearance                    (2)
```

In our example, the local test on the Architect's database tells us that we don't have to check the Owner's database for new bridge B4 because its clearance of 5 meters equals or exceeds the previously designed bridges. However, B5 must be checked globally because its clearance of 4 meters is lower than the previous bridges. Similarly in the Owner's database, truck T3 must be checked globally because its height is greater than that of truck T2.

5.2.4. Globally Verifiable Changes

Constraints cannot always be checked locally in response to potentially violating updates. We derive global queries, at compile time, that refer to all the relevant databases in order to verify the constraints in a configuration. For instance, for the case of insertions into relation Owner::Truck, the global test to detect constraint violations is:

```
Select   new_trucks.Truck_ID,  new_trucks.Height, Bridges.Bridge_ID
From     Architect::Bridges, Owner::new_trucks
Where    Bridges.Zone = new_trucks.Zone
   and   Bridges.Clearance < new_trucks.Height                         (3)
```

When we run the global test against our set of nonlocally verifiable changes, we discover that the pair of bridge B5 and truck T3 violate the constraint because the former has a clearance of 4 meters and the later has a height of 4.5 meters.

5.2.5. Violations and Notifications

Violations result from the constraint checking process, and they identify important design inconsistencies in the configuration. Often, participants may be notified of violations for constraints that were specified by other participants. Therefore, the notifications should have the relevant content for the participants to infer the cause of a constraint violation and the means to correct a violation. For instance, in our running example the owner should be notified of the trucks that are too high in order to pass under all the bridges in their service zone.

The contents of the violations are specified as a part of the constraint itself, and the attributes relevant to a violation are generated during the constraint checking process. That is, the contents of the violations are retrieved during constraint checking. The attributes are extracted by a query that is obtained by rewriting the global constraint checking query (refer Section 5.2.4). The query for computing the set of violations can be derived automatically for a subset of the class of general SQL queries using techniques described in [Ceri 85]. The violations received by each participant are stored in their violations database and can be accessed by interested participants when needed.

6. Conclusions

The problem of supporting civil construction has led us to a number of interesting questions that have not been emphasized in other engineering or scientific disciplines. The nature of the construction industry is such that a database for construction support must consist of a large number of independent databases. This situation contrasts with, say, a software development project or the design of an integrated circuit, where ownership is usually within a single company. Further, the construction industry is characterized by an unusual amount of interaction with the owner, many trial designs, and sequential modification of design components, especially as the structure is being built.

Fortunately, we can characterize many of the important requirements of a building using constraints. In particular, we have concentrated on constraints that affect data in two or more disciplines. Our model for change management includes the following features:

1. We support large trees of versions, each version coming from one of the many disciplines that constitute the design and construction team. Versions are stored by *deltas*, the changes from their parent configuration.

2. We support constraints as first-class components of the system. Since it is impossible for a change in one discipline's database to cause a change in some other database, the only way to coordinate design and construction is for constraints, and their violations, to become part of configurations. Violations are reported to appropriate participants, according to policies that are laid down when the constraint is asserted.

3. We compile constraints into local and global portions, attempting to check locally for new constraint violations where possible. We have developed a notion of local constraint checking to test that a change cannot create a new constraint violation by looking only at local data.

4. We support configurations comprising a version from each discipline, the constraints that pertain to the configuration, and the violations of those constraints. Violations are computed incrementally, and locally where possible, from the changes that led to the versions that participate in the configuration.

The combination of versions, configurations, and constraints provides a formal basis for the efficient management of changes in the AEC domain and other domains as well. There are many aspects of this combination that still need to be explored. For instance, we are continuing to investigate algorithms for discovering and using local constraint checks and to use incremental change information (deltas). We believe that this area will be a fruitful one for research in the years to come.

6.1. Implementation Status

The model of versions has been implemented on top of an ORACLE database management system using the Pro*C compiler to specify dynamic SQL links. The version manager stores the changes (deltas) that are made when going from a parent version to a new child version. Operators have been written to incorporate changes into active versions, to instantiate versions from deltas, and to compare hierarchically linked versions. The configuration manager is the next implementation step.

We initially implemented part of the constraint manager on the Starburst database system at IBM Almaden Research Center, because it provides good change monitoring facilities [Widom 89, 90]. In that environment, a set of interdisciplinary constraints was tested with a sample set of architectural, structural and mechanical databases for a commercial office building. We are finishing the constraint manager using ORACLE in order to integrate it with our version and configuration management software.

Acknowledgments

We gratefully acknowledge the support of the National Science Foundation through grant IRI-91-16646, and the IBM Almaden Research Center and the Stanford Center for Integrated Facility Engineering for the use of their computing facilities. We also thank Jennifer Widom for her input to this project and the Bechtel Group, Inc., for access to their project information during our constraint study. We also thank Marianne Siroker for her help in preparing this paper.

References

[Barbara 89] Barbara, D. and H. Garcia-Molina, "The Demarcation Protocol: A Technique for Maintaining Arithmetic Constraints in Distributed Database Systems," *Extending Database Technology Conference*, LNCS 580, pages 373--397, Vienna, March, 1992.

[Batory 85] Batory, D. and W. Kim, "Modeling Concepts for VLSI CAD Objects," *ACM Trans. Database Systems*, Vol. 10, No. 3, pp. 322-346, 1985.

[Blakeley 89] Blakeley, J. A. and Coburn, N. and Larson, P. "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," *ACM Transactions on Database Systems*, Vol. 14, No. 3, pp. 369-400, 1989.

[Bry 92] Bry, F., Manthey, R. and Martens, B., "Integrity Verification in Knowledge Bases," *Logic Programming*, LNAI 592 (subseries of LNCS), pp. 114-139, 1992.

[Ceri 85] Ceri, S. and Gottlob, G., "Translating SQL into Relational Algebra: Optimization, Semantics and Equivalence .of SQL Queries," *IEEE Transaction of Software Engineering*, Vol 11, No 4, pp 324-345, 1985.

[Ceri 90] Ceri, S. and Widom, J., "Deriving Production Rules for Constraint Maintenance," *Proceedings of Sixteenth International Conference on Very Large Data Bases*, pp. 566--577, 1990.

[Ecklund 87] Ecklund, D.J., E.F. Ecklund, R.O. Eifrig and F.M. Tonge, "DVSS: A Distributed Version Storage Server in CAD Applications," *Proceedings of the 13th VLDB Conference*, Brighton, England, pp. 269-274, September 1987.

[Gupta 92] Gupta, A. and Ullman, J.D., "Generalizing Conjunctive Query Containment for View Maintenance and Integrity Constraint Checking," *Workshop on Deductive Databases*, JICSLP, 1992.

[Gupta 93a] Gupta, A. and Tiwari, S., "Distributed Constraint Management for Collaborative Engineering Databases," *Proceedings of the Second International Conference on Information and Knowledge Management*, 1993.

[Gupta 93b] Gupta, A. and Widom, J., "Local Checking of Global Integrity Constraints," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49-59, 1993.

[Holtz 82] Holtz, N.M., *Symbolic Manipulation of Design Constraints - An Aid to Consistency Management*, Design Research Center, Carnegie Mellon University, DRC-02-012-82, 1982.

[Howard 86] Howard, H. C., and Rehak, D. R., *Interfacing Databases and Knowledge Based Systems for Structural Engineering Applications*, Technical Report EDRC-12-06-86, Engineering Design Research Center, Carnegie-Mellon University, Pittsburgh, PA, November 1986 (PhD dissertation).

[Howard 89a] Howard, H. C., Levitt, R. E., Paulson, B. C., Pohl, J. G., and Tatum, C. B., "Computer-Integration: Reducing Fragmentation in the AEC Industry,"*Journal of Computing in Civil Engineering,* Vol. 3, No. 1, pp. 18-32, 1989.

[Howard 89b] Howard, H. C., and Rehak, H. C., "KADBASE: A Prototype Expert System-Database Interface for Engineering Systems," *IEEE Expert,* Vol. 4, No. 3, pp. 65-76, Fall 1989.

[Katz 86] Katz, R.H., E. Chang and R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases," *Proceedings of the ACM SIGMOD Conference*, Washington D.C., pp. 379-386, May 1986.

[Katz 87] Katz, R. H. R., Bhateja, E., Chang, D. Gedye, and V. Trijanto, "Design Version Management," *IEEE Design and Test*, Vol 4, No.1, pp. 12-22, Feb. 1987.

[Katz 89] Katz, R. H., and E. Chang, "Inheritance Issues in Computer- Aided Design Databases," In *Object Oriented Database Systems*, K. Dittrich, and U. Dayal Eds., Springer-Verlag, Berlin, Germany, 1989.

[Katz 90] Katz, R. H., "Toward a Unifying Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, Vol. 22, No. 4, pp. 375-407, December 1990.

[Keller 94] Keller, Arthur M. and Ullman, Jeffrey D., "A Version Numbering Scheme with a Useful Lexicographical Order," submitted for publication, 1994.

[Ketabchi 87] Ketabchi, M.V. and V. Berzins, "Modeling and Managing CAD Databases," *IEEE Computer Magazine*, 20, 2, pp. 93-102, Feb. 1987.

[Klahold 86] Klahold, P., G. Schlageter and W. Wilkes, “A General Model for Version Management in Databases,” *Proceedings of the 12th VLDB Conference*, Kyoto, Japan, pp. 319-327, August 1986.

[Krishnamurthy 93] Krishnamurthy, K., *Version and Configuration Management for Collaborative Design*, CIFE Tech. Report No. 92, Stanford University, November 1993.

[Krishnamurthy 94a] Krishnamurthy, K., and Law, K., “Towards a Formal Model of Version and Configuration Management for Collaborative Engineering,” Submitted to the *Eight Annual Engineering Database Sympoisum*, Minneapolis, MN, September 1994.

[Krishnamurthy 94b] Krishnamurthy, K., and Law, K., “A Versioning and Configuration Scheme for Collaborative Engineering,” *Proceedings of First Congress on Computing in Civil Engineering*, ASCE, Washington, DC, June 1994.

[Landis 86] Landis, G.S., “Design Evolution and History in an Object Oriented CAD/CAM Database,” *Proceedings of the 31st COMPCON Conference*, San Francisco, pp. 297-305, March 1986.

[Law 90] Law, K.H., T. Barsalou, and G. Wiederhold, “Managing of Complex Structural Engineering Objects in a Relational Framework,” *Engineering with Computers*, Vol. 6, pp. 81-92, 1990.

[Leblang 84] Leblang, D.B. and R.P. Chase, “Computer Aided Software Engineering in a Distributed Workstation Environment,” *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Practical Software Development Environments*, pp 104-112, April 1984.

[Levy 93] Levy, A. and Sagiv,Y., “Queries Independent of Updates,” *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pp 171-181, 1993.

[Lorie 83] Lorie, R. and W. Plouffe, “Complex Objects and Their Use in Design Transactions,” *Proceedings of Databases for Engineering Applications, Database Week* 1983, pp 115-121.

[Nicolas 82] Nicolas, J.M., “Logic for Improving Integrity Checking in Relational Data Bases,” *Acta Informatica*, 18(3), pp. 227–253, 1982.

[Qian 88] Qian, Xiaolei, “Distributed Design of Integrity Constraints,” *Proceedings of the Second International Conference on Expert Database Systems*, The Benjamin/Cummings Publishing Company, pp. 205–226, 1988.

[Rasdorf 86] Rasdorf, William J. and Fenves, Steven J., “Constraint Enforcement in a Structural Design Data,” *Journal of Structural Engineering*, 112(12), pp. 2565–2577, 1986.

[Rochkind 75] Rochkind, M., “The Source Code Control System,” *IEEE Transactions on Software Engineering*, Vol SE-1, No. 4, pp. 364-370, December 1975.

[Sriram 92] Sriram, D., Ahmed, S. and Logcher, R., “A Transaction Management Framework for Collaborative Engineering,” *Engineering with Computers*, 8(4), 1992.

[Tiwari 93a] Tiwari, S., *Design Information Management in Complex Engineering Projects*, Technical Report, Job No. 90777, Research & Development, Bechtel Group, Inc., November 1993.

[Tiwari 93b] Tiwari, S., and Howard, H. C., “The Management of Design: A Design Notification Scheme for Distributed AEC Framework,” *First International Conference on the Management of Information Technology for Construction*, Singapore, August 1993.

[Tiwari 94] Tiwari, S., H.C. Howard, “Distributed AEC Databases for Collaborative Design,” *Journal of Engineering with Computers* (to appear), Springer-Verlag, 1994.

[Ullman 89] Ullman, J.D., *Principles of Database and Knowledge Base Systems*, Vol. 2, Computer Science Press, New York, 1989.

[Widom 89] Widom, J. and Finkelstein, S.J., *A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems*, IBM Research Report RJ 6880, IBM Almaden Research Center, Almaden, CA, 1989.

[Widom 90] Widom, J. and Finkelstein, S.J., “Set-oriented Production Rules in Relational Database Systems,” *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp.259-270, 1990.