Constraint Management On Distributed Design Databases *

Ashish Gupta
Department of Computer Science
Stanford University, CA 94305
agupta@cs.stanford.edu

Sanjai Tiwari
Center for Integrated Facility Engineering
Stanford University, CA 94305
tiwari@cive.stanford.edu

1 Introduction

Engineering design, such as building and aircraft design, involves many component specialties. For instance building design involves architects, structural designers, mechanical designers, electrical designers *etc.*. Each specialist has a domain-specific view of the design data, but needs to cooperate with other specialists throughout the design and construction phases of the building. The architect comes up with a plan for the building. The various designers work on the preliminary designs to satisfy the code and standards requirements. The contractor is responsible for executing the design. We refer to this collaboration between Architect, Engineers, and Contractors as the AEC framework [TH93]. Inconsistencies in design lead to expensive change orders and redesign, which in turn lead to delays and cost overruns for the project.

We propose the use of *integrity constraints* to specify design requirements and as a means of formally stating the interdependencies between various disciplines. We also describe the architecture of a distributed constraint management system (DCMS) that facilitates early detection of design inconsistencies via constraint checking. Consistency requirements may be imposed on a single site, or may involve multiple sites. That is, integrity constraints may either refer to a single database or refer to multiple databases. We refer to such constraints as *local* and *global* respectively, as illustrated by the following example.

Example 1: Consider a construction scenario and two relations from the contractor's database:

`owns`$(Crane\_id, Cost\_per\_hour)$ *% gives operating costs of the contractors cranes.*
`crane`$(Crane\_id, Floor\_id, Capacity)$ *% gives location and lifting capacity of cranes.*

Now consider a relation in the structural designer's database:

`column`$(Column\_id, Floor\_id, Weight)$ *% gives location, weight of concrete support columns.*

A *local* constraint C_l on the designer's database might require that every column on floors 4 and higher weigh less than 10 tons in order to avoid heavy structural elements on high elevations. A *global* constraint C_g might require every floor to have at least one crane that has the capacity to lift the heaviest column on that floor.

Not all the global design constraints are available at the time of database creation, instead they arise during the actual design process, *i.e.*, when the databases interact while running domain specific applications. Therefore, we need a facility for specifying and checking constraints that works independently of the underlying design data. Our proposed architecture for the Distributed Constraint Management System is designed to be built on top of existing databases. It treats constraints as first class objects,

*Work supported by NSF grants IRI-91-16646 and IRI-911668.

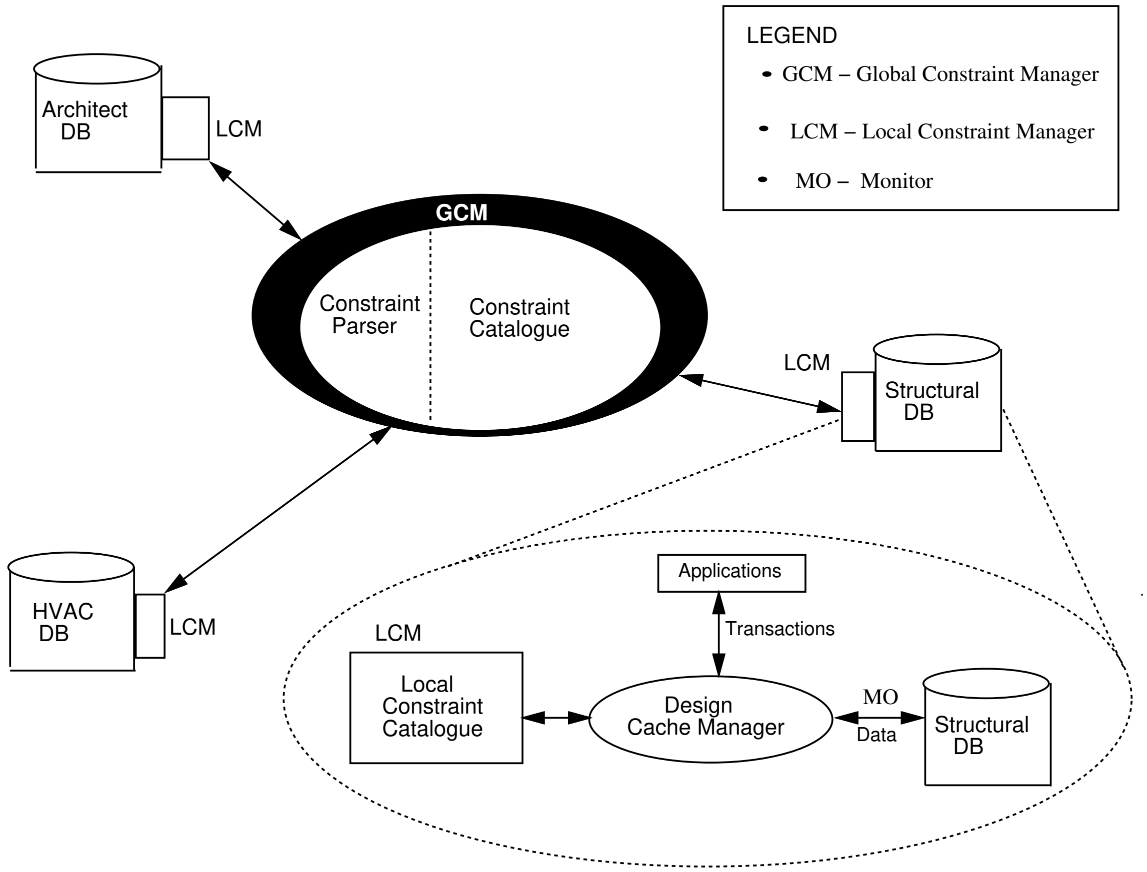

Figure 1: The Distributed Constraint Management System

and it allows constraints to be queried and updated because constraints are often as important as the data on which they are being imposed. The layered architecture of the DCMS emphasizes autonomy of individual databases. Our architecture tries to minimize the dependence on the functionality provided by the underlying databases, because different databases may provide radically different functionality.

Another dimension of the problem involves checking constraints when the underlying databases are updated. Checking global constraints can be very expensive due to the need to access remote data. Thus, compile time constraint optimizations are important for optimizing the run-time checking process. Our architecture is designed to incorporate many stages of optimizations. We consider *checking* constraints but not correcting their violations. Instead, we issue notifications to the participants affected by the constraint violations. A prototype of our architecture is being implemented on top of the ORACLE database system.

2 Architecture of DCMS

The proposed architecture for the *Distributed Constraint Management System* is outlined in Figure 1. Applications do domain specific reasoning and analysis and in the process alter the design database objects. The **design cache manager** at each site supports interaction of the applications with design databases. The cache manager allows checkin-checkout transactions and keeps track of the changes made to the design [KL94].

Constraints regulate the design changes made by applications. Constraints are specified, preprocessed and stored by the **Global Constraint Manager** (GCM). The GCM fragments constraints, derives optimization information, and distributes this derived information to the participating sites. At individual sites the **Local Constraint Managers** (LCM) are responsible for constraint checking. The compiled information sent to the LCMs is stored on the individual sites in **persistent catalogs**. Catalogs are also used by the GCM for global checking. Catalogs support efficient constraint checking by providing fast access to the compiled information at run-time when the GCM and LCMs use the information to check constraints. Catalogs also support query and update of the constraints. **Monitors** track changes to the objects in the design databases and detect the occurrence of operations that potentially violate constraints [CW90]. The LCM initiates the constraint checking process when prompted by a monitor.

3 Constraint Specification and Compilation

The constraint language used in our system is a variant of the language proposed in [CW90], extended to express constraints on multiple autonomous databases. Constraints are specified as inconsistent design states, *i.e.*, the condition that becomes true upon a constraint violation.

Example 2: Consider constraint C_g of Example 1. The violation condition for C_g is expressed as follows. Designer::`Columns` refers to the `Columns` relation on the Designer's site.

Designer::`Columns`.*Weight* $\geq$ all (Select `Cranes`.*Capacity* From Contractor::`Cranes`
Where `Cranes`.*Floor_Id* = `Columns`.*Floor_Id*)
Actions: Notify(Designer, Contractor, Project Manager);

The above specification says that constraint C_g is violated if there is a column whose weight is greater than the capacity of every crane on that floor. If such columns exist, the designer, contractor, and project manager should be notified.

The constraint specification language is declarative and has many advantages over a procedural specification language. It allows a uniform high-level representation of global constraints and avoids ad-hoc user-programmed constraints that rely on the varying semantics of the underlying systems. As a consequence, the designer does not need to know the internal details of how individual sites manage integrity constraints. Uniform representation also facilitates a high-level interface to the constraint repository for constraint querying and update. Declarative specifications also facilitate extensive compile-time optimizations [Ull89].

3.1 Constraint Compilation

From the high-level specification of constraints, the compilation phase extracts the information needed for constraint checking. A constraint is syntactically parsed to form its parse tree that is used to derive the query that evaluates the constraint violation condition. The parse tree is also used to derive the set of invalidating operations, *i.e.*, the operations on the database objects that might violate a constraint [CW90]. For instance, the sample constraint C_g is compiled to obtain the following query and invalidating operations.

Example 3: Query to evaluate the constraint

Define `Bad_Set` as: (Select * From Designer::`Columns`
Where *Weight* $\geq$ all (Select `Crane`.*Capacity* From Contractor::`Cranes`
Where `Cranes`.*Floor_Id* = `Columns`.*Floor_Id*))

Action: Notify(Designer, Contractor, Project Manager) using contents of `Bad_Set`;

Bad_Set contains the columns whose weight exceeds the capacity of all the cranes assigned to the corresponding floor. *Actions* specify the action to be taken on a constraint violation.

The set of invalidating operations that cause the above query to be computed:
Designer Site: Insert(Designer::column), update(column.$Weight$), update(column.$Floor\_Id$).
Contractor Site: Delete(Contractor::cranes), update(cranes.$Capacity$), update(cranes.$Floor\_Id$).

The invalidating operations are monitored at the local databases. Often a cache manager provides the monitoring facility for design databases. The set of invalidating operations can be optimized in many ways. For instance, if the floor_id of the columns is non-updateable (columns cannot be moved from floor to floor) then there is no need to monitor such an operation in Example 3.

Another optimization strategy we have devised and implemented is *Local Checking* [GSUW93]. E.g., consider constraint C_g and suppose the designer adds a new column col_1 of weight 10 tons on floor fl_1. We need to ensure that there is a crane on floor fl_1 that can lift column col_1. Suppose floor fl_1 already had a column weighing 12 tons. Assuming that C_g was satisfied before adding col_1, we can infer that floor fl_1 has a crane with capacity of at least 12 tons and thus adding col_1 of weight 10 tons will not violate C_g. Note, the remote relation *crane* on the contractors site was not read for the above check. The compilation phase derives local tests and stores them in the LCM catalogs at various sites. Other constraint checking optimizations strategies can also be incorporated at compile time (as in [Nic82, KSS87, LST87, BMM92]).

Compilation also fragments global constraint into database specific local components that can be evaluated efficiently with fewer run-time translation overheads. For instance, some of the data and schema translations needed to check a global constraint can be compiled into the local fragments produced for the individual sites (unit conversions like inches to cms, or mapping a beam in the designer's database to a girder in the contractor's database). Fragmentation also reduces the amount of information that local sites need to communicate to the GCM for constraint checking. The constraint compilation process is described in detail in [TH93].

4 Constraint Checking and Implementation

Constraint checking is the most expensive phase of constraint management and potentially is needed every time a participating database is updated. Monitors on the database objects inform the cache manager of the updates that can potentially violate any constraint. The cache manager computes net changes made to the database and thereby avoids redundant checks; for example, a deleted object that is reinserted should not be considered as having changed. Only the relevant changes are sent to the LCM. A more detailed discussion of the change management issue can be found in [Hal91, KL94].

The LCM checks each constraint against the relevant updates by running the local tests stored in the constraint catalog at that site. If the local tests fail, then the GCM is informed about the updates. A global query – stored in the global constraint catalog – is initiated in order to check the constraint globally. Example 3 shows the global query for constraint C_g. The database updates are used to instantiate and restrict this global query as much as possible in order to make the global query efficient [Nic82, KSS87, LST87, BMM92]. For instance, if column col_1 is added to floor fl_1, then only the cranes on fl_1 need to be checked and not all cranes in the building. Distributed query optimization techniques can be used to execute the global query produced by the GCM [CP84, OV91].

4.1 Notifications

Notification is the final phase of constraint management, activated only if a constraint is violated. Often notifications need to be issued to selected participants who are responsible for resolving the design inconsistency arising from the constraint violation. For instance, if constraint C_g is violated due to adding a column, the contractor may be responsible for allocating a crane with sufficient capacity; the designer need not be notified about this constraint violation. This asymmetric participation of sites can be modeled by listing the participants in order of their responsibility in the constraint specification. The notifications could be broadcast to all the sites referred by the constraint if a notification list is not provided.

4.2 Implementation

A prototype constraint manager has been implemented in the Starburst database system at IBM-Almaden Research center and currently is being ported to ORACLE. The GCM uses *lex* and *yacc* to parse constraints and to derive compile time information like invalidating operations and local checks. Constraint catalogs are maintained as database tables and can be queried using SQL. The GCM and LCMs are implemented in C++ and C and they interact with the underlying database (and catalogs) using Starburst API and ORACLE OCI and ProC. Our architecture is general enough to be ported on top of other commercial database systems. Production rules [CW90] and triggers are used to monitor the database operations at run-time (in Starburst and ORACLE respectively). The current prototype uses a single site project database.

5 Future Work

Many interesting directions have emerged from our research. One issue is to keep track of design conflicts (constraint violations) that are detected and further track those that have been resolved or are in the process of being resolved. Thus conflicts that do not receive any attention for a given time period can be detected and reminders issued to participants. We are also considering a more general checkin-checkout mechanism for design data and a design cache-manager that would provide the LCM with a set of changes made to the design. There is also a need for allowing *what-if* changes to be made – a sort of pseudo commit process – which will be very useful for iterative design.

References

[BMM92] F. Bry, R. Manthey, and B. Martens. Integrity Verification in Knowledge Bases. *LNAI 592 (subseries of LNCS)*, 114–139, 1992.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. *VLDB*, 1990.

[GSUW93] A. Gupta, S. Sagiv, J. D. Ullman, and J. Widom. Constraint Checking with Partial Information. PODS 1994.

[KL94] Karthik Krishnamurthy and Kincho Law. A Versioning and Configuration Scheme for Collaborative Engineering. In *First Congress on Computing in Civil Engineering, Washington, D.C.* ASCE, 1994.

[Hal91] K. Hall. *A Framework for Change Management in a Design Database.* PhD thesis, Stanford University, Department of Computer Science, (report number STAN-CS-91-1379), 1991.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. *VLDB*, 1987.

[LST87] J.W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity Constraint Checking in Stratified Databases. *Journal of Logic Programming*, 4(4):331–343, 1987.

[Nic82] J. M. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982.

[OV91] T. M. Oszu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[SAL92] D. Sriram, S. Ahmed, and R. Logcher. A Transaction Management Framework for Collaborative Engineering. *Engineering with Computers*, 8(4), 1992.

[TH93] S. Tiwari and H. C. Howard. Distributed AEC Databases for Collaborative Design. *The Journal of Engineering with Computers*", Springer-Verlag, 1994.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems 1 and 2.* CSP, 1989.