Distributed Constraint Management for Collaborative Engineering Databases *

Ashish Gupta † Sanjai Tiwari ‡

Abstract

The engineering design process, such as airplane or building design, involves the participation of many independent specialists who need to share information stored in several autonomous design databases. To insure consistency of the final design, high-level constraints need to be defined and enforced on the design objects stored in these preexisting databases. The constraints need to be verified when the participating databases are updated. In this paper, we present an architecture of a distributed constraint management system that checks design constraints on preexisting autonomous databases. The architecture emphasizes independence of constraint management from the underlying databases systems to avoid compromising site autonomy.

To provide this independence, we present a declarative constraint language for specifying constraints, which facilitates compile time optimizations such as constraint fragmentation and local validation strategies. We also support set-oriented updates that model transactions in engineering applications, and notifications that model interaction between designers. Early detection of constraint violation and appropriate notifications to the participants can avoid expensive project delays and cost overruns. A prototype of the proposed architecture is being implemented using the Starburst database system at IBM Almaden Research Center.

Keywords: collaborative engineering, cooperation in design databases, integrity constraints, knowledge sharing

*This work is supported by NSF grants IRI-91-16646 and IRI-911668.

†Department of Computer Science, Stanford University, CA 94305, USA. email: agupta@cs.stanford.edu

‡Center for Integrated Facility Engineering, Stanford University, Stanford, CA 94305, USA.
email: tiwari@cive.stanford.edu

1 Introduction

With the increasing complexity and exacting demands on nearly every kind of engineering product, it has become critical to assist the design process with automation. Databases are one such tool available to designers in order to help them store, retrieve, and interpret massive amounts of information. Collaborative design involves many participants who may use domain-specific databases. Even though the databases are autonomous, they store related information. Integrity constraints are a useful way of specifying and checking these relationships between design databases.

In general, integrity constraints can either refer to the information in just one database or refer to information stored in multiple databases. We shall refer to such constraints as *local* and *global* respectively, as illustrated by the following example.

EXAMPLE 1.1 Consider a construction site where a contractor and a designer are two of the participants. Consider two relations stored in the contractor's database:

R_1: $\texttt{owns}(Crane\_id, Cost\_per\_hour)$.
R_2: $\texttt{crane}(Crane\_id, Floor\_id, Capacity)$.

Relation R_1 has information about the machinery owned by the contractor and the machinery's operating costs. Relation R_2 has information about the locations at which machinery is being used on the site. Some of the machinery being used may not be owned by the contractor. Now consider a relation in the designer's database:

R_3: $\texttt{column}(Column\_id, Floor\_id, Weight)$.

Relation R_3 has information about the weight and location of the concrete columns that are being used in the building.

The contractor can have a local constraint C_l that says that every crane that is used on a floor higher than the fourth must be owned by the contractor because of the high liability associated with using rented cranes on higher elevations. Such a constraint refers to data only on the contractor's site and is therefore local.

There can also be a global constraint C_g that requires every floor to have at least one crane that has the capacity to lift the heaviest column on that floor. This constraint refers to both the contractor's and the designer's database. □

Not all the global design constraints are available at the time of database design, but arise during the actual design process, *i.e.*, when the databases interact while running domain specific applications. A facility for specifying and checking constraints that works independently of the underlying design data is therefore useful. This paper discusses the architecture of a constraint management system that is geared to be built on top of existing databases. We propose a layered architecture that emphasizes the autonomy of individual databases. Each site has a local constraint manager (LCM) for managing constraints locally, and there is a central global constraint manager (GCM) for global validation. Later, we explain how this separation facilitates optimization. Both the LCM and GCM are add-on layers that need some basic functionality from the underlying database. Our architecture tries to minimize the support that the database needs to provide, because the databases used by different designers may provide radically different functionality.

When the underlying databases are updated, constraints may be violated. Checking global constraints can be very expensive because they involve remote data access and distributed transactions. Therefore, compile time constraint optimizations become very important to reduce run-time overheads. In our architecture, constraints are compiled at the GCM because the GCM maintains a constraint repository and has more information than any of the LCMs at the individual sites. We fragment global constraints across multiple sites, and derive local checks, which are distributed to the LCMs for locally checking global constraints. For instance, constraints that use data purely from one site are identified in the compile phase. These optimizations increase the amount of local run time checking and reduce the global communication required to check constraints.

All the information generated at compile time is stored in persistent catalogs to be used at run-time. The catalog at the GCM stores information such as the global constraint specification, notifications, and invalidating operations. The catalogs at the LCMs store information such as the local fragments, local tests, and the invalidating operations relevant to that site. The GCM also maintains a constraint repository for the constraints in the system. The constraint repository is a very important part of the system because constraints represent semantic connections between the underlying databases. These connections change (*i.e.*, constraints are added and deleted) as the design progresses. Constraints should therefore be managed independent of the underlying data. The project participants should be able to query, update, and alter the constraints: our architecture supports such a constraint repository.

In our opinion, enforcing constraints over autonomous databases is not a realistic option in the design domain. Therefore, we consider the problem of *checking* for constraint violations, not correcting them; however, we issue notifications to the participants affected by the constraint violations. A prototype of our architecture is being implemented using Starburst database system [HFLP89].

1.1 Background Work

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. 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 and it exploits data distribution to infer optimizations that cannot be done at any of the local sites. We now discuss some of the existing work on integrity checking and how it can be used with our architecture.

Ceri and Widom [CW90] 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 and part of the functioning of the LCM at any of the underlying sites. In fact, our implementation uses their system as the backend.

[Qia88] describes techniques for distributing global constraints between sites in a way that reduces communication at run time. These techniques can be used at the preprocessing phase at the GCM. Similarly the optimizations that result from instantiating constraints by updates made to the database can be used by both local constraint managers and the global constraint manager. Such optimizations are discussed in [Nic82, KSS87, LST87, SV86, BMM92]. Distributed constraint checking can also be made more efficient by using the demarcation protocol [BGM92] or by generating queries that are sufficient to allow us to infer that a constraint has not been violated [BBC80, BCL89, Elk90, GU92]. In Section 4.2 we discuss how these ideas are used in our system.

Of the existing prototypes in the engineering domain, the Dice project [SAL92] addresses the coordination and communication problems of multiple design transactions on a single project database. Kadbase [HR85] addresses the problem of schema integration and data communication between multiple engineering databases. The key factors that distinguish our research from the previous work in the collaborative design domain are local autonomy and efficient techniques for maintaining constraints on distributed databases.

1.2 Outline of Paper

In the next section, we provide a brief overview of the application domain used in this paper, namely the building design domain. Section 3 presents the proposed system architecture. Section 4 describes constraint compilation, validation steps, notifications, and the constraint repository. Implementation of the prototype system is described in Section 5. Conclusions and future work appear in Section 6. We present sample constraints from the building design domain in Appendix A.

2 Collaborative Design Framework

Engineering design, such as building and aircraft design, involves many component specialities and subspecialities. A building design project can range from construction of multistory structures and apartment complexes to air-hangers and dams. The building design project involves the participation of many designers: architects, structural designers, mechanical designers, electrical designers, and contractors. Similar observations hold for other design domains; for instance, aircraft design involves participation of avionics, instrumentation, structural and propulsion designers. In this paper, we use the building design scenario as a representative domain.

Each specialist has a domain-specific view of the design data, but needs to cooperate throughout the design and construction phases of the building. The owner of the building has certain key functional requirements and he or she may hire an architect, who comes up with several plans and associated costs for design and construction. Once a plan has been approved by the owner, the designers – structural, mechanical and electrical – work on the preliminary designs to satisfy the code and standards requirements for the building. We will refer to this collaboration between Architect, Engineers and Contractors as the AEC framwork [TH93]. Generally, the architect takes the overall lead in the design process, which involves the cooperation of the participating designers throughout the project duration. Inconsistencies in the design lead to expensive change orders and redesign, which in turn lead to delays and cost overruns for the project. Examples of constraints that insure consistent design in the AEC domain are given in Appendix A. The next section presents an architecture of a distributed constraint management system (DCMS).

3 Architecture of DCMS

Some of the important issues that deserve special attention in the collaborative design domain are (1) autonomy of domain-specific databases; (2) effective and efficient interaction between databases; and (3) validation of the consistency of the designs stored in these databases. The first issue is important because the databases are maintained by different designers, who may not want other participants to modify their data for proprietary and legal reasons. The second issue concerns the efficiency and ease of the design process, while the third issue deals with the overall consistency of the final design. Early identification of design conflicts can avoid readjustments in designs and result in fewer change orders and rework during the construction phase. The proposed architecture helps in maintaining the consistency of the design efficiently, while not compromising the site autonomy. Figure 3 outlines the architecture of the Distributed Constraint Management System explained below.

- **Application:** A program that needs input data for domain specific reasoning and analysis. The output of an application, e.g. structural analysis and material cost estimation, usually involves processing design attributes of the database objects.

- **Constraint:** A constraint in our system refers to a violation condition expressed over multiple design databases. Note that the constraints in Example 1.1 and in Appendix A are the conditions that the database needs to *satisfy* in order for it to be consistent. For purpose of clarity, we use the term constraints to refer to both the violation condition and assertions when the context makes the usage unambiguous. However, in the DCMS a constraint is specified strictly as a violation condition.

- **Global Constraint Manager (GCM):** The global constraint manager is responsible for storing and preprocessing constraints. During constraint preprocessing, a global constraint is partitioned into a set of local constraints based on the sites that store the design objects referred to in that constraint. The preprocessing phase also derives optimizations that are then stored in persistent catalogs. The global constraint manager keeps a repository of all design constraints, and it provides facilities for updating and querying constraints. The GCM also coordinates the various local constraints managers.

- **Constraint Catalogs:** Catalogs support efficient constraint validation by providing fast access to the information derived by the GCM. The catalog entries are determined at constraint compile time, and these

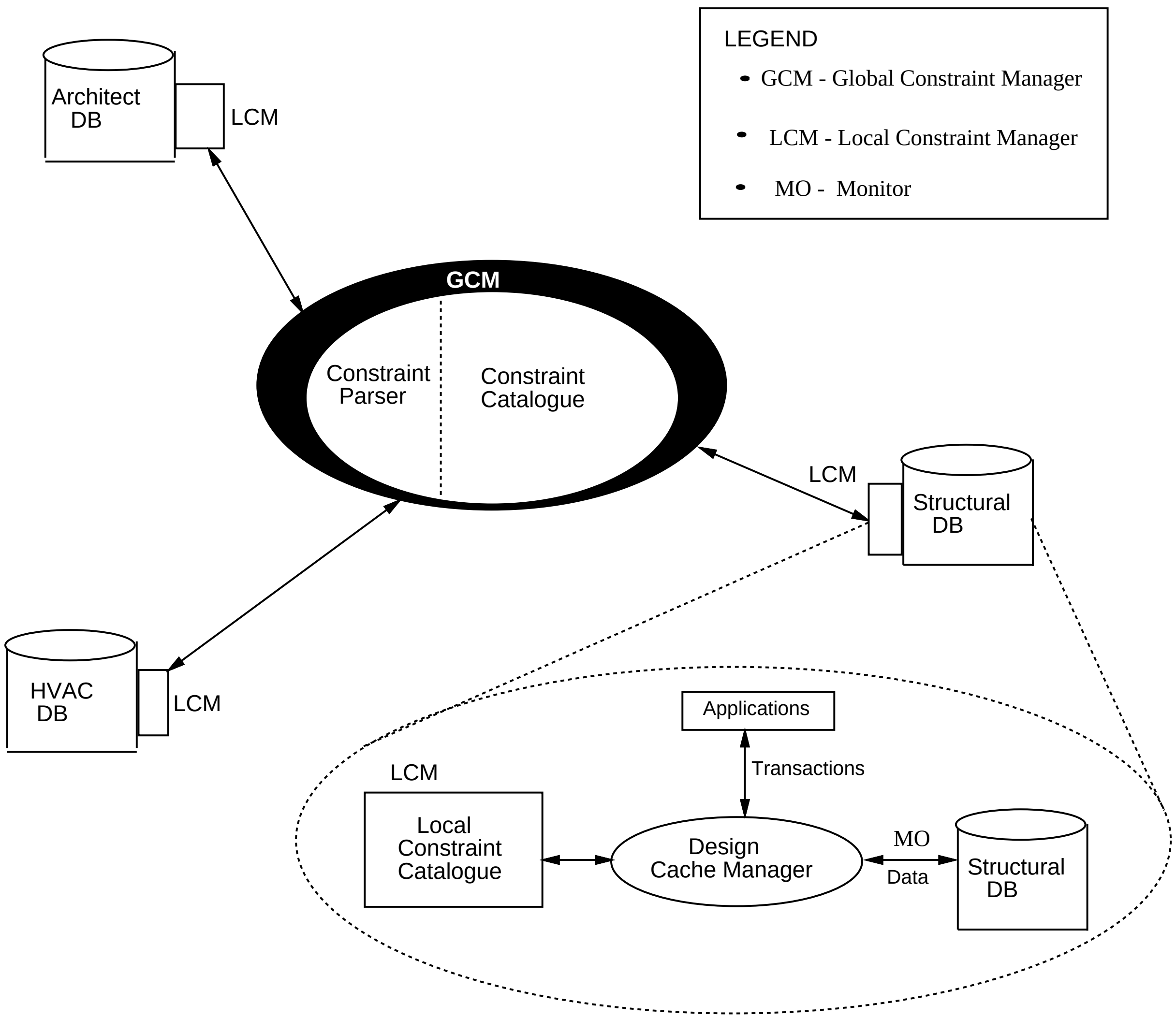

Figure 1: The Distributed AEC Framework consists of Design Databases, Global and Local Constraint Managers, Design Cache Managers that monitor changes to the local databases, and applications that run on the different databases.

entries are used by the global and local constraint managers for constraint validation at run-time.

- **Local Constraint Manager (LCM):** The local constraint manager at a site manages fragments of global constraints that are relevant to that site. The primary function of an LCM is to initiate constraint checking when data on its database is updated. The LCM uses the optimizations described later to make constraint checking efficient. In case it is not able to locally check constraints, it informs the GCM of the updates which then completes the checking process.
- **Monitors:** Monitors keep track of changes to the objects in a design database. A monitor informs the LCM when an invalidating operation occurs. An invalidating operations is an operation that can potentially violate a constraint.
- **Design Cache Manager:** A design cache manager supports interaction of applications with design database. It provides a mechanism for checkin-checkout transactions and keeps track of the changes made to the design.

The constraint manager can use the underlying database for manipulating the information needed for constraint checking, *i.e.*, catalogs can be stored as persistent tables in the database. This approach not only benefits from the facilities provided by the DBMS, but also makes it possible to support a *constraint repository* that is useful for managing constraints. For instance, a designer may need to query the repository for existing constraints on an object before specifying another constraint. In addition, the whole architecture is geared to support set-oriented changes, *i.e.*, constraints that are evaluated on sets of changes made to the database – a very useful facility for design transactions.

4 Constraint Preprocessing

The constraint preprocessing phase infers optimization information to be used at run-time for constraint validation. The subsequent sections describe the stages of constraint processing. We discuss a high-level SQL-like language for constraint specifications in Section 4.1. Constraint compilation steps that result in constraint catalog entries for that constraint are described in Section 4.2. Run-time validation using these catalogs is presented in Section 4.3, and Sections 4.4 and 4.5 discuss extensions to support notifications and the constraint repository.

4.1 Constraint Specification

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 specificed as inconsistent design states, *i.e.*, the condition that becomes true upon a constraint violation. In a collaborative design scenario it is more efficient and natural to specify inconsistent states than enumerate all possible consistent states.

EXAMPLE 4.1 Consider constraint C_g of Example 1.1 that required every floor to have at least one crane that has the capacity to lift the heaviest column on that floor. The violation condition for C_g is expressed as follows:

Designer::`Columns`.Weight $\geq$ all
  (**select** `Cranes`.Capacity
  from Contractor::`Cranes`
  where `Cranes`.Floor_Id = `Columns`.Floor_Id)
  actions:
   Notify(Designer, Contractor, Project Manager);

Designer::`Columns` refers to the `Columns` relation on the Designer's site. The above statement says that constraint C_g is violated if there is some column whose weight is greater than the capacity of every crane on that floor. In case of such a violation, the concerned participants, *i.e.*, designer, contractor, and project manager, should be notified. □

The language is declarative and gives us the following advantages over a procedural specification language:

- **Uniform Representation:** A uniform high-level representation of global constraints avoids ad-hoc user-programmed constraints that rely on the varying semantics of the underlying systems. Avoiding this low-level dependence allows the designers to specify "what" constraints need to be enforced rather than "how" constraints should be enforced. As a consequence, the designer does not need to know the internal details about how the individual sites manage integrity constraints. Uniform representation also facilitates a high-level interface to the constraint repository, *i.e.*, constraints can be queried and updated.
- **Optimization:** A declarative representation of constraints allows many compile-time optimizations. Site-specific constraint fragments and local integrity checks can be derived from the global specification. These optimizations will be cumbersome (often not possible [Ull89]) to derive from ad-hoc procedural constraint specifications.

4.2 Constraint Compilation

From the high-level specification of constraints, the compilation phase extracts the information needed for constraint validation. A constraint is syntactically parsed to form its parse tree [ASU86], which is used to derive a procedural specification. The parse tree is used to derive the set of invalidating operations, *i.e.*, the operations on the database objects that might violate a constraint [CW90]. The sample constraint C_g is transformed to the following procedural specification containing invalidating operations on the local database sites.

EXAMPLE 4.2

Create **Constraint::Columns-Cranes**
when
/ when any of these invalidating operations occur */*
Designer Site
 updated $Designer :: Columns.Weight$,
 updated $Designer :: Columns.Floor\_Id$,
 inserted *into* $Designer :: Columns$
Contractor Site
 updated $Contractor :: Cranes.Capacity$,
 updated $Contractor :: Cranes.Floor\_Id$,
 deleted *from* $Contractor :: Cranes$

/ check if there are any tuples in the Conflict Set */*
if exists Conflict_Set:
  (**select** * from Designer::Columns
   where Weight $\geq$ all
    (**select** Crane_Capacity
    from Contractor::Cranes
    where Cranes.Floor_Id = Columns.Floor_Id))
/ User Specified Actions */*
then select Floor_Id, Column_Id, Weight from Conflict_Set;
  Notify(Designer, Contractor, Project Manager);

The **when** clause of the above production specifies a set of potentially invalidating operations at the designer and the contractor sites. When any of these invalidating operations occur, the **Conflict_Set** may need to be computed to determine the columns whose weight exceeds the capacity of all the cranes assigned to the corresponding floor. If required, the conflict set is computed by the SQL query. The **then** clause specifies the action to be executed if the constraint is violated. □

The invalidating operations are monitored at the local databases. For instance, a particular database system may provide triggers that provide this monitoring. Often the cache manager will provide the monitoring facility for design databases. An entry for every monitor is made in the local catalogs (see Figure 2) corresponding to every invalidating operation relevant to that site. An optimization that results from distributing monitors to the local DBMS is that some of the invalidating operations can be eliminated using attribute-level checks, like range restriction or unique attribute checks. For instance, if the floor_id of the columns is non-updatable (the columns cannot be moved from floor to floor) then there is no need to monitor such an operation and it can be removed from the **when** clause in 4.2. Thus, LCMs go through an extra phase of optimization in order to minimize the set of invalidating operations that need to be monitored.

Local Tests

As illustrated in Example 4.2, a constraint violation condition can be evaluated as a global query on multiple databases. After an update, brute-force validation may involve reading data from multiple sites. Instead, local tests only use the information available at the site of the update to determine if the global constraint is actually violated. The compilation phase derives local tests for every invalidating operation and stores them in the local catalog of the corresponding site (Figure 2). Local tests can be derived for a large class of constraints using conjunctive query containment techniques [GU92]. Related techniques are also discussed in [Nic82, KSS87, LST87, BCL89, BGM92, BMM92] Local checks are a very useful optimization in the distributed scenario because they often avoid the remote communications associated with global validation [GW93]. For instance, suppose the designer adds a new column col_1 of weight 10 tons on floor fl_1. In order to ensure that there is a crane on floor fl_1 that can lift column col_1, we would need to remotely read the contractor's database. However, suppose floor fl_1 already had a column whose weight was 12 tons. Assuming that the constraint C_g was satisfied before adding col_1, we can infer that floor fl_1 has a crane with capacity more than 12 tons.

EXAMPLE 4.3 The catalogs in Figure 2 are for the sample constraint C_g. The entries in the global constraint catalog are indexed by the unique constraint ID, and contain information needed for the global validation. The more interesting entries appear in the local constraint catalogs which are distributed to the local sites. Each entry corresponds to a monitor for a constraint, a test for local validation, and a local query whose results are sent to the GCM if the local tests fails. The global catalogs also contains information about the notifications to be delivered to the appropriate sites. □

In summary, the constraint preprocessing phase produces procedural specifications and run-time optimizations from a declarative constraint specification. Besides the advantages discussed above, fragmentation of a global constraint also produces 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 validation.

4.3 Constraint Validation

Constraint validation is the most expensive phase of constraint management and is needed every time a local database is updated. Figure 4.3 shows the chronological order of the steps that may be executed during a typical constraint validation phase at a particular database. The design cache manager assists in retrieval and update of design objects from the database as shown in step ①. Monitors on the database objects inform the cache manager of the changes that can potentially violate any constraint. Step ②corresponds to the initiation of constraint validation by the design cache manager. The cache manager computes net changes made to the database and therefore 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].

In step ③the LCM validates each constraint against the changes by running the local tests stored in the constraint catalog at that site. Local tests use only the information available in the changed database to infer whether the changes violate the affected constraints. Such optimizations are useful if the accessing remote data is more expensive than local processing. As shown in Figure 2, these tests are compiled and stored for every potentially invalidating operation.

Global Constraint Catalog					
Constr_Id	Constraint Specification	Sites Referred	Global Query to compute conflict set	Notification	...
C_g	Designer:: Columns.Weight ≥ all (select Cranes.Capacity from Contractor:: Cranes where Cranes.Floor_Id = Columns.Floor_Id)	Designer Contractor	select * from Designer::Columns where Weight ≥ all (select Crane_Capacity from Contractor::Cranes where Cranes.Floor_Id = Columns.Floor_Id)	select * from Conflict_Set Notify Designer, Contractor, Project Manager	...
⋮	⋮	⋮	⋮	⋮	

Local Constraint Catalog @ Designer			
Constr-Id	Monitors	Local Tests	To GCM
C_g	insert into Columns	**select** * **from** Columns C, Inserted-Columns IC **where** C.Floor_Id = IC.Floor_Id **and** C.Weight ≥ IC.Weight	**select** * **from** Inserted-Columns
C_g	update Columns	**select** * **from** Columns C, Updated-Columns UC **where** C.Floor_Id = UC.Floor_Id **and** C.Weight ≥ UC.Weight	**select** * **from** Updated-Columns
⋮	⋮	⋮	⋮

Figure 2: Catalogs produced on compiling constraint C from Example 1.1

In case the local test fails, then the GCM needs to be informed about the relevant updates, as shown in step 4. A global query – stored in the global constraint catalog – is initiated in order to evaluate the constraint violation condition. The local updates are used to restrict this global query as much as possible in order to make the global query less general and therefore more efficient [Nic82, KSS87, LST87, BMM92]. For instance, if column col_1 is added to floor fl_1, then we need to verify constraint C_g only for fl_1 and not all the floors in the building. To avoid inferring how to use the local updates at run time, the constraint catalog at the GCM are built at the compile time itself. Distributed query optimization techniques can be used to execute the global query produced by the GCM [CP84, OV91]. The GCM constraint catalog also stores the query needed to generate the notifications (see Figure 2).

4.4 Notifications

Notification is the final phase of constraint management, activated only if a constraint violation has been found. If the notification list is not provided in the constraint specification, the notifications are broadcast to all the sites referred to in a constraint. The notification message should contain sufficient information to determine the cause of a violation. Too much or too little information can be misleading; if redundant or incomplete information is presented to the designer, he or she may have to sort through the notification message or request additional information. The conflict set contains the information needed to generate notification messages, *i.e.*, it contains the design objects that violate a constraint.

Often notifications need to be issued to selected participants who may be responsible for resolving the design inconsistency arising from the constraint violation. For instance, when a column weighing 12 tons is added to floor fl_1, the contractor may be responsible for allocating cranes with sufficient capacity; the designer need not be notified about this constraint violation. This asymmetric participation of sites is modeled by specifying an order for notifying the participants in the notification list. The order can reflect the extent of a participant's responsibility for resolving a given conflict.

4.5 Constraint Management Summary

We started with a high-level definition of a constraint and compiled it into a detailed procedural program in terms of design objects and operations on them at different databases. This global program was split into site-specific local constraint fragments that are stored in local catalogs. The LCM is invoked by the local cache manager, which monitors the invalidating operations at

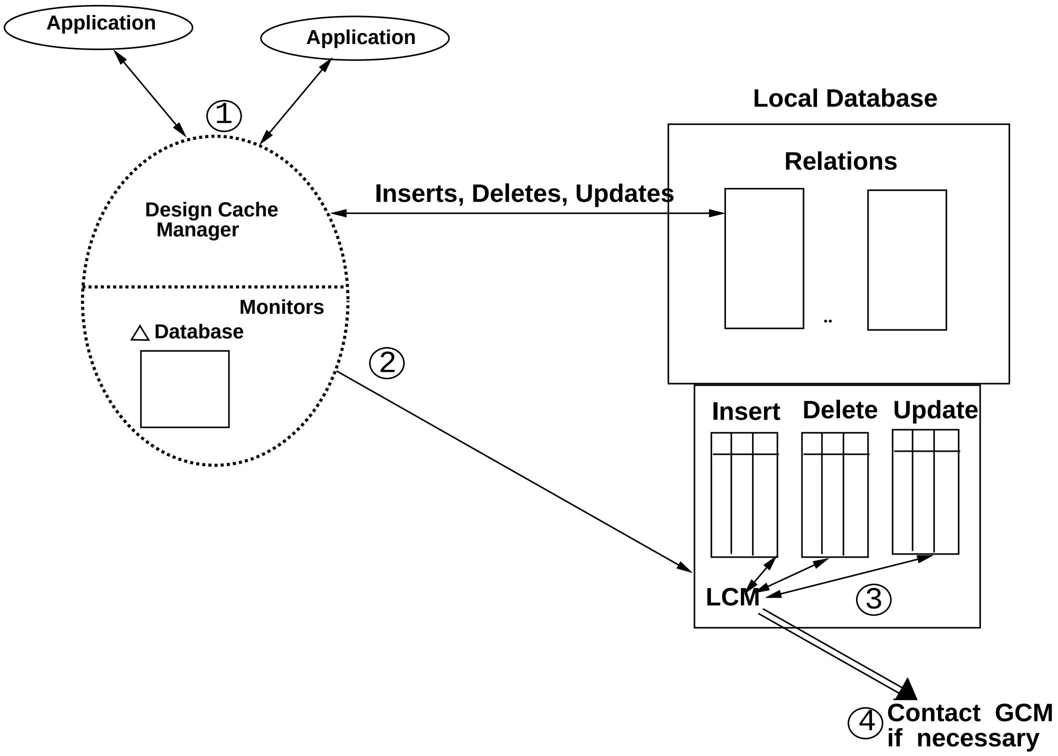

Figure 3: A simplified run-time picture of the constraint management process. The accumulated changes for the transactions are monitored and sent to the LCM. The LCM attempts to do a local verification before going through a global verification process.

the respective sites. A local validation of constraints is activated using the local tests from the catalog, followed by a global validation only if necessary. Notification may be issued to selected participants if a constraint violation is detected.

Constraint Repository

In a design scenario, the constraints are as important as the design objects on which they are enforced. Constraints help capture the design decisions for a project, which can be useful not only during project design, but also during service and maintenance of the product. Thus, design constraints should be treated as first class database objects in the sense that designers should be able to query and update these constraints just like any other database object. The catalogs used for constraint maintenance can also be used as a part of constraint repository. A constraint repository should facilitate easy specification of constraints on the underlying databases and allow prioritizing or disabling constraints. A designer may request information of the following type:

- Which constraints will be affected if object X is updated?
- Find all the design objects that participate in constraints involving object X.

Such queries can be easily answered using the DBMS query language if the constraint catalogs are stored as objects in the database. A repository can also be used to answer "what-if" questions, which are extremely useful during the early design phases. For instance, a designer may want to know the consequences of adding columns weighing 15 tons on floor fl_1 without actually changing the database.

5 Implementation

A prototype constraint manager has been implemented in the Starburst database system at IBM-Almaden Research center. The GCM uses *lex* and *yacc* to parse a given constraint specification and derive the set of invalidating operations on the databases. The parsing rules use the attribute-inheritance approach to differentiate a given constraint specification [ASU86] with respect to the objects referred to in that constraint. Constraint catalogs are maintained as database tables and can be queried using Starburst-SQL. C++ is used to implement the global and local constraint managers that interact with the constraint catalogs using Starburst API on IBM RS6000 workstation. However, our architecture is general enough to be ported on top of commercial database systems like Oracle and Sybase.

EXAMPLE 5.1 Starburst production for monitoring updates to the weight of the structural column at the designer's database:

```
Create rule monitor.colum-wghts on designer::columns,
   when updated(Columns.Weight),
   then 'insert into monitor.updates
         select "Tx001_Designer", "Columns",
                "Weight", ou.Weight, nu.Weight
         from ou as (old_updated()),
              nu as (new_updated())
         where ou.Column_Id = nu.Column_Id';
```

The syntax of rules of the type stated above is explained in [WF90]. □

Production rules [CW90] are used to monitor the database operations at run-time to inform the LCM about relevant updates in a transaction (as illustrated in the above example). As future work, we are considering a more general checkin-checkout mechanism: a design cache-manager that would provide the LCM with a set of changes made to the design. The cache-manager would supplement the repository in allowing *what-if* changes to be made – a sort of pseudo commit process – which will be very useful for iterative design. Currently, the catalogs provide part of the repository's functionality: the ability to answer querys on constraints. Intelligent notification and the remaining functionality of constraint repository need to be added to the current implementation. The current prototype simulates distributed communication using a single site project databases on Starburst database system.

6 Conclusions and Future Work

In this paper, we presented an architecture for managing constraints that involve distributed, autonomous design databases. We emphasized autonomy of the participating sites and gave many compile time optimizations to enable efficient constraint checking. In particular, we presented constraint fragmentation, distribution, and local validation strategies. Most of the existing work in constraint checking can be incorporated into our system due to its modular nature. Notifications are an important aspect of a system that manages design constraints. The system generates notifications when constraints are violated. We also proposed a constraint repository as a useful tool in managing constraints themselves, which can be very large in number and whose specification may change as the design evolves. Implementation of a prototype is in progress using the Starburst Database System.

However, there are many open issues that are not addressed in this paper and provide directions for future work. One such issue involves keeping track of the design conflicts (constraint violations) that have been resolved or are in the process of being resolved. The system can monitor the conflicts that do not receive any attention for a given time period and can send reminders to the participant and their supervisors. Such a framework assists in capturing the organizational aspect of the design process since coordination often occurs at multiple levels. Notifications may be stored in a *notfication database* to allow the system to track when and to whom notifications were issued and how many of the design conflicts have been resolved. A notification database can assist in keeping the log or history of all the conflicts, which can be later queried to get useful information about the status of the design at a given point in time. Such information will be required not only for the duration of the project but also during service, maintenance, and retrofit of the product.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[BGM92] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Arithmetic Constraints in Distributed Database Systems. In *Extending Database Technology Conference, LNCS 580*, pages 373–397, Vienna, March, 1992.

[BCL89] J. A. Blakeley, N. Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[BBC80] P. A. Bernstein, B. T. Blaustein, and E. M. Clarke. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. In *Proceedings of the Sixth Conference on Very Large Data Bases*, pages 126–136, 1980.

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

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill Book Company, New York, N.Y., 1984.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proceedings of Sixteenth International Conference on Very Large Data Bases*, pages 566–577, 1990.

[Elk90] C. Elkan. Independence of Logic Database Queries and Updates. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 154–160, Nashville, TN, June 1990. ACM SIGACT-SIGMOD-SIGART.

[GU92] Ashish Gupta and Jeffrey D. Ullman. Generalizing Conjunctive Query Containment for View Maintenance and Integrity Constraint Checking. In *Workshop on Deductive Databases, JICSLP*, 1992.

[GW93] Ashish Gupta and Jennifer Widom. Local Checking of Global Integrity Constraints . In *Proceedings of the ACM SIGMOD, International Conference on Management of Data*, 1993.

[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.

[HFLP89] L. M. Hass, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–388, 1989.

[HR85] H. C. Howard and H. C. Rehak. Kadbase: A Prototype Expert System-Database Interface for Engineering Systems. *IEEE Expert*, 4(3):169–181, 1985.

[KSS87] Robert Kowalski, Fariba Sadri, and Paul Soper. Integrity Checking in Deductive Databases. In *Proceedings of the Thirteenth International Conference on Very Large Databases (VLDB)*, pages 61–69, 1987.

[LJD91] Ray E. Levitt, Yan Jin, and Clive L. Dym. Knowledge Based Support for Management of Concurrent, Multidisciplinary Design. *AI in Engineering, Design and Manufacturing*, 2(5):77–95, 1991.

[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.

[MH91] David V. Morse and Chris Hendrickson. Model for Communication in Automated Interactive Engineering Design. *Journal of Computing in Civil Engineering*, 5(1):4–24, 1991.

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

[OV91] Tamer M. Oszu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Qia88] Xiaolei Qian. Distributed design of integrity constraints. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 205–226. The Benjamin/Cummings Publishing Company, 1988.

[SV86] E. Simon and P. Valduriez. Integrity Control in Distributed Database Systems. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 622–632, 1986.

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

[TH93] Sanjai Tiwari and H. C. Howard. Constraint Management on Distributed AEC Databases. In *Fifth International Conference on Computing in Civil and Building Engineering*, pages 1147-1154, ASCE, 1993.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, Volumes 1 and 2. Computer Science Press, New York, 1989.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270. ACM, 1990.

A Sample Constraints from Building Design

- The capacity of a crane used at the construction site should be greater than the maximum weight of the building columns it has to lift.

- The constraint:
  Floor Elevation $\geq 1.10 * ($Ceiling_Ht. + Beam_Depth + HVAC_Duct_Depth$)$
  should be satisfied under all circumstances. Notify all professionals in case of conflict.

- An exterior wall should not be deleted by the architect if reference to the wall exist in designer's and contractor's databases.

- If the size of a window is increased by more than 20% in designer's database, then notify the HVAC engineer about possible change in the size of the air-conditioning duct for that room.

- If wall thickness is changed in the architect's database, notify the structural designer for possible change of depth of the supporting beams.

- The size of the reinforcement bar (bar#) used by the contractor should be within 10% range as specified in the structural designer's database.

- The availability and cost of the formwork for round columns dictates that their diameter be even number in the range of 12"-32" and there should be at least twelve such columns in the contractor's database.