Constraint Management on Distributed Design Configurations

Sanjai Tiwari
Department of Civil Engineering
Stanford University
Stanford, CA 94305-4020
tiwari@cive.stanford.edu

Ashish Gupta
Department of Computer Science
Stanford University
Stanford, CA 94305-2140
agupta@cs.stanford.edu

Submitted to *Journal of Engineering with Computers*

Abstract

Architecture, Engineering and Construction (AEC) design processes involve the participation of many designers who may be working independently in geographically distinct locations. Independent designs evolve due to changes made by the designers and discipline-specific CAD applications, and these changes need to be evaluated and configured with respect to a set of project constraints for overall design consistency. In addition, each designer needs to be informed about the relevant changes made by other designers. Most rework orders result from inadequate sharing of the evolving design information between various participants in the design process. The awareness of important changes through automated constraint management can avoid expensive reworks at the later stages of the projects.

In this paper, we will present an incremental approach to constraint checking in AEC design configurations. A configuration represents a design state stored in multiple databases: each configuration has a set of *discipline-specific databases* and an associated set of *inter-disciplinary constraints*. The constraints are evaluated on the databases to give the third component of configurations: a set of *violations*. Including violations in the configuration definition allows us to support partially consistent configurations. Partially consistent configurations are important from a practical standpoint since all the design information may not be available, or may not be consistent, at all stages of the AEC design process. We will provide a formal description for configurations and the semantics of changes on configurations. This framework facilitates incremental constraint checking and allows us to support the notions of *persistent* and *what-if* configurations. We also provide a classification of constraints that is helpful in selective evaluation of constraints.

1 Introduction

In a collaborative design environment, different participants need to share information about the critical aspects of their design data and the changes that occur in their design. The changes result from fine tuning of design and reevaluation of the design goals as the design progresses. Therefore, it is necessary to be able to handle the evolution of discipline-specific designs and be able to systematically cope with changes throughout the design process. Each AEC design configuration deals with a large amount of design data which may be distributed over many discipline-specific databases. The participants designs are assumed to be stored in a database or some electronic format which can be logically queried and examined for the most recent design changes.

The participants must also check the consistency of their design periodically with respect to a set of project constraints. The project 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 from different disciplines such as architectural, structural engineering, mechanical, electrical and plumbing (MEP) [TH93]. 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.

A design configuration consists of a collection of the individual design databases and a set of inter-disciplinary constraints. Participants can publish their databases to be a part of a design configuration, *i.e.*, a configuration evolves as new databases are published by the participating designers or new constraints are introduced in a configuration. We will define a configuration in terms of its component databases, constraints, and constraint violations, and provide a formal semantics for handling changes on different design databases. The constraints are treated as independent entities and not associated with a particular database. Treating constraints independently of the underlying design data enables reusability of constraints both across projects and in different stages of the same project.

Another important problem we address in this paper is that of inconsistent design configurations. It is seldom the case that all the constraints in a project are satisfied and, therefore, the constraint violations are also stored as an important part of a configuration definition. The violations found as a result of the constraint checking process on configurations are notified to appropriate participants [THL93]. We also define *persistent* and *what-if* configurations: persistent configurations are baselined at major milestones in a project, such as facility design reviews. What-if configurations are temporary configurations that evaluate the effect of intermediate design changes in between persistent configurations. The "what-if" facility allows designers to evaluate the impact of possible changes without actually making the changes.

It is imperative that the design constraints be managed efficiently since there can be thousands of constraints in a project. We present a system that evaluates constraints only for the most recent design changes that have occurred since the last persistent configuration. That is, constraints are evaluated incrementally with respect to a configuration in order to avoid redundant constraint checking. 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 use of a high-level constraint language allows many compile-time optimizations which use information about the distribution of design data between different databases. One such language and related optimizations were presented in [TH93]. Runtime optimizations can be used to avoid unnecessary checks and data transfer [GW93]. The underlying databases are used to store the optimization information derived at constraint compile time, in *constraint repositories* that are stored as database objects. At run-time, the derived information can be obtained by simply looking-up the repositories.

1.1 Paper Outline

In Section 2, we give a high level description of a design configuration followed by its mathematical formulation in terms of databases, constraints and violations. Section 3 gives an overview of

Figure 1: Project Configuration

constraint management using changes in a configuration. Violations and notifications resulting from the constraint checking process are explained in Section 4. Section 5 discusses how constraint repositories assist in answering queries about project constraints and how the repositories are implmentated as database objects. Related work and conclusions appear in Section 6 and Section 7, respectively.

2 Configurations

In this section, we formally describe configurations for collaborative design databases and explain the different ways in which configurations can evolve during the design process. Section 2.1 disusses a high-level description of a configuration and its components. Section 2.2 defines a formal model for configurations such that the model captures the distributed nature of design configurations. Finally, Section 2.3 describes changes to the component databases and constraints, and how they relate to overall configuration development.

2.1 Configuration Model

A configuration $\mathcal{CF}$ is defined on a set of discipline-specific databases involved in a project. Figure 1 illustrates a project configuration consisting of discipline-specific databases, such as structural, mechanical, and electrical, and inter-disciplinary constraints. As motivated in Section 1, the databases can have constraints defined on them and these constraints are also part of the configuration. Finally, the configuration also includes the set of *violations*, *i.e.* those objects in the databases that violate some constraint. In this section we consider a representative constraint from the AEC domain and highlight some of the characteristics of constraints and violations that need to be captured while formalizing the notion of a configuration.

EXAMPLE 2.1 Consider a configuration where information about the building columns is stored in a structural designer's database, and information about the cranes that are assigned to lift these columns is stored in a contractor's database. Now consider a constraint C_{eg} on this configuration which requires that every floor be assigned at least one crane that has the capacity to lift the heaviest column on that floor. The constraint is violated if the weight of some column is more than the capacity of every crane on that floor. We use an extension of SQL database language to express the constraint C_{eg} as follows [CW90, TH93]:

```
Designer:: Columns.Weight ≥ all
        (Select   Cranes.Capacity
         From     Contractor:: Cranes
         Where    Cranes.Floor_Id =  Columns.Floor_Id );

        Notify(Designer, Contractor, Project Manager)
```

Designer:: Columns refers to the Columns objects in the Designer's database. The above specification says that constraint C_{eg} is violated if there is some column whose weight is greater than the capacity of all the cranes assigned to that floor. In case of such a violation, the concerned participants, *i.e.*, designer, contractor, and project manager, should be notified. □

The above constraint may be violated if the designer introduces a new column on a floor, or if the contractor removes some existing crane. In general, for any constraint expressed in the language, it is possible to identify the database operations that could potentially violate that constraint expressed in the constraint language [CW90]. For constraint C_{eg} in Example 2.1, the potentially violating operations are as follows:

EXAMPLE 2.2

$$\left.\begin{array}{l}\textbf{updated}\ \text{Designer :: Columns.Weight,}\\ \textbf{updated}\ \text{Designer :: Columns.Floor\_Id,}\\ \textbf{inserted}\ \text{into Designer :: Columns,}\end{array}\right\}\ \text{Designer Database}$$

$$\left.\begin{array}{l}\textbf{updated}\ \text{Contractor :: Cranes.Capacity,}\\ \textbf{updated}\ \text{Contractor :: Cranes.Floor\_Id,}\\ \textbf{deleted}\ \text{from Contractor :: Cranes,}\end{array}\right\}\ \text{Contractor Database}$$

Each invalidating operation is specific to either the Designer's or the Contractor's database. The Designer need not be aware of the Contractor's invalidating operations and vice-versa. Therefore, the set of invalidating operations can be fragmented and distributed between the two databases [TH93]. In addition to invalidating operations, other information used in constraint checking can also be inferred from the constraint specification, and distributed among the participating databases [GT93]. We discuss the constraint fragmentation and distribution process in more detail in Section 3. □

Note that the specification of C_{eg} also mentioned the participants who need to be notified in case the constraint was violated. The content of the notification is very crucial to constraint checking because the actions taken by the participants are determined by the information they receive. For instance, in Example 2.2, if there is a column that cannot be lifted by any crane, then the participants need to be notified of the offending column and probably of the capacity of the cranes for that floor. One or more of these participants may want to store the violation notification they receive, and take appropriate action at a later time. Therefore, violations need to be distributed to the relevant discipline-specific databases so that they can be queried and used later. Violations are discussed in more detail in Section 4.

The above discussion motivates the need for a configuration model to capture the distributed nature of its components. In the next section, we formalize the definition of a configuration in terms of its components and their fragments. Some of the components were motivated above and some will be described in later sections.

2.2 Definitions

A configuration $\mathcal{CF}$ is defined on a set of discipline-specific databases $\{DB^1, \ldots, DB^n\}$. The configuration is physically distributed into two parts:

- Inter-disciplinary information, that is shared by all the disciplines and is stored in the global constraint repository.
- Discipline-specific components, such as invalidating operations, constraint and violation fragments, that are maintained locally at the participating databases in local constraint repositories.

In this paper, we use the terms discipline and participant inter-changeably. In general, a discipline can have more than one databases participating in a configuration.

2.2.1 Interdisciplinary Information

In the equations below, superscripts are used to refer to particular databases and subscripts are used for enumeration (for instance, O_i^j refers to the i^{th} object O stored in database DB^j).

$$\begin{aligned} \mathcal{CF} &= [\ \mathcal{D},\ \mathcal{C},\ \mathcal{V}\ ] &&(1)\\ \text{where } \mathcal{D} &= \text{a set of databases; one from each discipline participating in } \mathcal{CF}\\ \mathcal{C} &= \text{a set of constraints associated with } \mathcal{CF}\\ \mathcal{V} &= \text{a set of constraint violations for } \mathcal{CF} \end{aligned}$$

$$\begin{aligned} \mathcal{D} &= \{DB^1,\ DB^2,\ \ldots,\ DB^n\} &&(2)\\ \text{where } n &= \text{number of databases participating in } \mathcal{CF} \end{aligned}$$

$$\begin{aligned} \mathcal{C} &= \{C_1, \ldots, C_m\} &&(3)\\ \text{where } m &= \text{number of constraints associated with } \mathcal{CF}\\ \text{where } C_i &= <C_{id},\ GCF,\ LCF> &&(4)\\ \text{and } C_{id} &: \text{a constraint identifier} \end{aligned}$$

$$
\begin{array}{rcl}
GCF & : & \text{a global constraint fragment} \\
LCF & : & \text{local constraint fragments distributed to participating databases}
\end{array}
$$

$$
\begin{array}{rcl}
GCF & = & \{<C_{id},\ GT_1,\ NT_1>,\ \ldots,<C_{id},\ GT_k,\ NT_k>\} \\
GT_i & : & \text{global test for checking a constraint} \\
 & & \text{with respect to } i^{th} \text{ invalidating update operation} \\
NT_i & : & \text{notification to be sent when a violation is detected by } GT_i
\end{array} \tag{5}
$$

$$
\begin{array}{rcl}
LCF & = & \{LCF^1,\ LCF^2,\ \ldots,\ LCF^n\} \\
 & & \quad \text{where } LCF^i \text{ is a local constraint fragment distributed to } DB^i
\end{array} \tag{6}
$$

$$
\begin{array}{rcl}
\mathcal{V} & = & \{\mathcal{V}^1,\ \mathcal{V}^2,\ \ldots,\ \mathcal{V}^n\} \\
 & & \quad \text{where } \mathcal{V}^i \text{ is the violation fragment of } \mathcal{V} \text{ stored at } DB^i
\end{array} \tag{7}
$$

Equation 1 defines a configuration to be a set containing sets of databases, constraints, and violations. Note that $\mathcal{V}$ can be computed from $\mathcal{D}$ and $\mathcal{C}$ using the procedure explained in Section 3. Equations 2 and 3 explain the components of a configuration in terms of their subcomponents. Equations 4 to 7 describe the components in terms of their fragments. They also capture the distributed nature of a configuration. Equation 2 says that the configuration needs to have one database from each of the n participating disciplines. Equation 3 defines the set of constraints imposed on $\mathcal{CF}$. Each constraint has an identifier, C_{id}, and associated global and local fragments (Equation 4). The global constraint fragment (GCF) needs to be shared between all n databases and contains a global validation test and a notification statement that is issued if the constraint is violated (Equation 5). Note that in general there can be a different global test and associated notification for every invalidating operation of a constraint. The GCF is logically stored in the global constraint repository as proposed in [THL93], but could physically be distributed across many sites. Local constraint fragments, on the other hand, contain database-specific information and are distributed to participating databases (Equation 6). Similarly, Equation 7 says that fragments of the violation $\mathcal{V}$ distributed to the local databases such that $\mathcal{V}^i$ contains the inconsistent object(s) that are of interest to the participant at database DB^i. A particular violation may be of interest to multiple participants, and would therefore appear in multiple fragments. As discussed in Section 4, we will simplify this general model to store each violation at exactly one database by associating additional properties with constraints and violations.

2.2.2 Discipline-Specific Information

We now describe the details of discipline-specific components maintained at the individual databases: namely constraint and violation fragments.

Constraint Fragments

LCF^m is the local constraint fragment of C_{id} stored at database DB^m:

$$
\begin{aligned}
LCF^m &= \{<C_{id},\ MO_1,\ LT_1>,\ \ldots,<C_{id},\ MO_k,\ LT_k>\} && (8)\\
\text{where } MO_i &: \text{ monitor for tracking the } i^{th} \text{ invalidating update at } DB^m\\
LT_i &: \text{ test for locally validating constraint } C_{id}\\
&\quad\ \text{when monitor } MO_i \text{ is triggered}
\end{aligned}
$$

$$
\begin{aligned}
MO_i &= <O,\ Op> && (9)\\
&\quad\ \text{a trigger for detecting operation } Op \text{ on database object } O
\end{aligned}
$$

$$
\begin{aligned}
O &\in DB^m && (10)\\
Op &\in \{I,\ D,\ U\} && (11)\\
\text{where } I &: \text{ an insert operation}\\
D &: \text{ a delete operation}\\
U &: \text{ an update operation}
\end{aligned}
$$

Equations 8 to 11 describe the database-specific characteristics of a constraint. Equation 8 describes the constraint fragment at the m^{th} database for constraint C_{id}. This local constraint fragment contains the constraint identifier and a set of monitors and local tests. The monitors detect the occurrence of operation Op (Equation 9) on a database object O if operation Op can potentially violate constraint C_{id}. For instance, Example 2.2 illustrates that many operations could potentially violate constraint C_{eg} of Example 2.1. Therefore, for that one constraint, many operations need to be monitored at the different discipline databases that participate in the configuration. The local test LT_i uses the information available at database DB^m in order to check for constraint violation (Equation 8). Equation 11 restricts the types of operations to be either insert, delete or update.

Violation Fragments

The set of violations $\mathcal{V}$ is fragmented across the participating databases such that $\mathcal{V}^m$ contains those violations that refer to the objects stored in DB^m (Equation 12).

$$
\begin{aligned}
\mathcal{V}^m &= \{<C_1,\ \mathcal{K}_1>,\ldots,<C_l,\ \mathcal{K}_l>\} && (12)\\
C_i &: \text{ an identifier for the violated constraint}\\
\mathcal{K}_i &: \text{ identifier (key) of the particular object O that violates the constraint } C_i
\end{aligned}
$$

$\mathcal{V}^m$ stores the identifiers of all the instances of the objects that cause violations, along with the identifiers of the constraints that object violates.

The violations are useful from many perspectives. They allow participants to issue queries about the inconsistent design objects and take corrective actions. Keeping the violations as part of the configuration helps in keeping track of the history of notifications issued to the participants, which is especially useful for administrative book keeping. We are also considering the problem of how to automatically update the set of violations when corrective action is taken; *i.e.* every time a violation is corrected in a new configuration, the effect of the corrective action is reflected in the set of violations. Updating the set of violations will help the participants keep track of the violations they have corrected.

2.3 Changes on Configurations

A configuration can change when either one of its constituent databases changes or when constraints are inserted or deleted from a configuration. We use $\Delta\mathcal{C}$ to refer to change in constraints and $\Delta\mathcal{D}$ to refer to change in the constituent databases. $\bigtriangledown$ is used to represent an operator or a "method" that returns the corresponding Δ for its input arguments. [Kri93] discusses how to determine $\bigtriangledown$.

$$\begin{aligned}
\text{Initial Configuration} \quad \mathcal{CF}_i &= [\ \mathcal{D}_i,\ \mathcal{C}_i,\ \mathcal{V}_i\ ] && (13)\\
\text{New Configuration} \quad \mathcal{CF}_j &= [\ \mathcal{D}_j,\ \mathcal{C}_j,\ \mathcal{V}_j\ ] && (14)\\
\text{Changes to constraints} \quad \Delta\mathcal{C} &= \bigtriangledown\ (\mathcal{C}_j,\ \mathcal{C}_i) && (15)\\
\text{Changes to databases} \quad \Delta\mathcal{D} &= [\ \Delta\mathcal{D}^1,\ \Delta\mathcal{D}^2,\ \ldots,\ \Delta\mathcal{D}^n\ ] && (16)\\
\text{where} \quad \Delta\mathcal{D}^m &= \bigtriangledown\ (\mathcal{D}_j^m,\ \mathcal{D}_i^m) && (17)
\end{aligned}$$

$\mathcal{CF}_i$ represents an initial persistent configuration (Equation 13). Equation 14 declares a new persistent configuration in terms of updated set of databases $\mathcal{D}_j$, and an updated set of constraints $\mathcal{C}_j$. The changes to a configuration consist of the changes made to constraints ($\Delta\mathcal{C}$ defined by Equation 15), and the changes made to the participating databases ($\Delta\mathcal{D}$ defined by Equation 16). The changes to the set of violations are computed implicitly when the constraint checking process computes the new set of violations $\mathcal{V}_j$.

3 Incremental Constraint Checking on Configurations

When a new configuration is created, the constraints need to be re-evaluated and a new set of violations needs to be computed. As described earlier, a new configuration is created when either one of its constituent databases changes or when constraints are added or removed from the configuration. 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 only consider the *changes* made to the configuration and do not evaluate all the constraints on the entire databases belonging to a new configuration.

In this section we describe how constraint checking progresses in a collaborative design environment. Section 3.1 describes two different types of configurations which are treated differently from the perspective of constraint checking. The two kind of configurations model two important scenarios that frequently arise in engineering projects. Section 3.2 gives a high-level description of how constraints are actually checked on these configurations.

3.1 Persistent and What-if Configurations

We distinguish between two types of configurations that may result due to changes made to the component databases.

Figure 2: Persistent Configuration

- **Persistent Configurations:** At certain stages in a project, participants may want to publish their databases in a new persistent configuration that is shared by all the project participants. At this stage, all the project constraints will be evaluated on the configuration databases and a new violations database is created to baseline the current persistent configuration. The creation of persistent configuration is shown in Figure 2. All the responsible designers are notified about the violations they are interested in. The set of violations are fragmented across the databases and stored as a part of the configuration.

- **What-If Configurations:** In between the persistent configurations, different designers may want to create their own private configurations to evaluate their latest design changes with respect to the other participants' design. These configurations cannot be declared persistent since the designer may decide to abort such changes later in the design process. The constraint checking for a what-if version is done with respect to some persistent configuration, *i.e.*, database changes are computed with respect to the last persistent configuration and only the creator of a what-if configuration is notified of the violations. Because the what-if

Figure 3: What-if Configuration

configuration is not persistent, the set of violations are not stored.

Figure 3 illustrates how changes at a given database DB^i can be used for constraint evaluation via a temporary "what-if" configuration.

Recall that the change to a configuration could be the addition of constraints resulting in a new set of constraints, probably more detailed and extensive than those in the earlier configuration. The new constraints are evaluated with respect to the databases belonging to the new configuration.

3.2 Constraint Checking

Constraint checking within a configuration involves three steps. (1) specification, (2) compilation, and (3) run-time checking.

Specification Constraint specification was introduced in Section 2, and additional examples are included in Appendix A. The 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. The constraint language used in our system is a variant of the language proposed in [CW90], extended to express constraints on multiple autonomous databases. We have also extended the language to specify the participant who creates the constraint, the participant responsible for violations, the content of the notifications, and the participants to be notified.

Compilation Even though constraint checking is done 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. For instance Example 2.2 illustrates the set of invalidating operations derived by constraint compilation. The constraint is also fragmented into discipline-specific components that can be evaluated at individual databases. In [TH93], we described the compilation process in detail. Constraint compilation also produces the tests that are used to check for constraint violations. For the constraint of Example 2.2, the test that checks for constraint violations is described in the following example.

EXAMPLE 3.1 If the test represented by following SQL query returns an answer then the constraint is violated.

```
Select  Columns.Id
From    Designer::Columns
Where   Weight ≥ all
        (Select  Crane_Capacity
         From    Contractor::Cranes
         Where   Cranes.Floor_Id = Columns.Floor_Id )
```

□

Note that the query in Example 3.1 refers to both the Designers and the Contractor's databases, *i.e.*, it refers to multiple databases. Therefore, the query is referred to as a *global query* and is stored as part of the Global Constraint Fragment. Sometimes it is also possible to check for a constraint violation without evaluating a global query that refers to multiple databases.

EXAMPLE 3.2 Suppose the structural designer adds a new column col_1 of weight 4 tons on floor fl_1 subject to Constraint C_{eg} from Example 2.2. In order to ensure that there is a crane on floor fl_1 that can lift column col_1, the structural designer would need to remotely access the contractor's database. However, suppose floor fl_1 already had a column whose weight was 6 tons. Assuming that there was no violation for constraint C_{eg} before adding col_1, we can infer that floor fl_1 has a crane with capacity at least 6 tons. Therefore, the 4 ton column cannot violate C_{eg}. □

We refer to this optimization as *local checking* [GW93]. The compilation phase derives the local tests for invalidating operation [GW93] whenever possible.

In summary, constraint compilation produces procedural specifications and run-time optimizations from a high-level constraint specification. Compilation also fragments a global constraint to produce 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 databases (unit conversions like inches to cms, or mapping a beam in the designer's database to a girder in the contractor's database [How91]). Fragmentation also reduces the amount of information that the local databases needs to send over the network for global constraint checking. The information produced by the compilation phase (invalidating operations, local tests, global tests, etc.) is stored in local and global constraint repositories that are described in Section 5. These repositories can be queried efficiently for the information derived at compile time when constraint checking takes place.

Checking Constraint checking on a configuration can be initiated whenever either the set of constraints change ($\Delta\mathcal{C}$) or the databases change ($\Delta\mathcal{D}$). Evaluating the constraints in $\Delta\mathcal{C}$ requires evaluating the new constraints against the databases in the new configuration. Evaluating constraints on $\Delta\mathcal{D}$ requires checking that no constraint in the new configuration is violated by the data in $\Delta\mathcal{D}$. In either case, the set of violations may change. The incremental constraint checking strategy works also when the change to the configuration includes changes to both constraints and databases. If the new configuration is persistent, then the new set constraint violations replaces the old set of violations. In case the configuration is a what-if configuration, the new set of violations is not stored.

Local tests are used only in the case that the underlying database changes ($\Delta\mathcal{D}$). If the local tests fail, then the global test is used (refer Figure 5 and Figure 6). For instance, when any of the invalidating operations listed in Example 2.2 occur and the corresponding local tests fail to validate the constraint, the query in Example 3.1 is evaluated to find the columns that are too heavy to be lifted by the available cranes. Local checking is a useful optimization technique in the distributed scenario because they often avoid the remote communications associated with a global constraint validation. A high-level description of the algorithms for checking constraints is presented in Appendix B.

4 Notification of Violations

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 structural designer should be notified of the columns that are too heavy to be lifted by the cranes along with the maximum limit of the available crane's capacity. The content 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. The query for computing the set of violations can be automatically derived for subset of the class of general SQL queries using techniques described in [CG85]. Appendix A illustrates constraint specifications that specify the contents of the violation and also have the appropriate queries to extract the required attributes. The violations received by each participant are stored in their violations database and can be accessed by interested participants when needed.

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 through which the participants can handle the constraint violations and their notifications more efficiently. For the purpose of classifying constraints in distributed configurations, we assign additional attributes to the components of a configuration: database objects, constraints and violations. 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:

Constraint->Source	Violation->Responsibility	Object->Owner(s)	Constraint->Type
X	X	X	**Local-Internal**
X	X	X&Y	**Global-Internal**
X	Y	Y	**Local-External**
X	Y	X&Y	**Global-External**

Figure 4: A Constraint Classification Scheme to handle Notifications

Figure 4 classifies constraints along two dimensions: local vs. global and internal vs. external. X and Y denote particular disciplines/participants. *Local constraints* involve data from only one database (X) as compared to *global constraints* that involve data from multiple databases (X&Y and possibly more participants). *Internal constraints* are created by the same discipline that is responsible for correcting the violations. By contrast, an *external constraint* is specified by a discipline that is not the same as the discipline responsible for correcting the violations. Note that there are a total of 6 possible rows in the table in Figure 4 after selecting the source of the constraint (X): 3 choices for the object owners (X,Y,$X\&Y$) and 2 for the discpline responsible for the violation (X,Y). However, the discipline responsible for violations has to own an object participating in the constraint. 2 of the 6 combinations do not satisfy this requirement: $<X, X, Y>$ and $<X, Y, X>$, and therefore do not appear in the table.

We make the local-global distinction since global constraints, involving multiple remote databases, are usually more time consuming to check than the local ones which involve no remote data access. In additional participants are usually more familiar with the constraints internal to the discipline than the external constraints imposed by participants belonging to different disciplines. Thus, in a what-if configuration a participant may first check the local constraints within the discipline, and then move on to global constraints.

EXAMPLE 4.1 We present example for each of the above category of constraints below, their detailed specifications are provided in appendix A. The object owners are implicit in the constraint specifications.

1. *Check that all the door frames for the personnel rooms in the architectural database are made of fiberglass.*
   Constraint->Source: Architect Violation->Responsibility:Architect
   Constraint->Type: Local-Internal

2. *Check that all the HVAC equipments in the mechanical database are connected to the electrical grounds in the electrical database.*
   Constraint->Source: Electrical Violation->Responsibility: Electrical
   Constraint->Type: Global-Internal

3. *Check that there are no conduits below 7'2" in the mechanical database.*
   Constraint->Source: safety_engineer Violation->Responsibility: Mechanical

Constraint->Type: Local-External

4. *Check that all the cranes in the contractor's database have capacity greater than the weight of the columns in the structural database that it has to lift.*
   Constraint->Source: Project Manager Violation->Responsibility: Structural
   Constraint->Type: Global-External

□

This classification also helps in fragmenting and distributing the constraint violations among the participating databases in order to ensure efficient retrieval of violations. In our simplified model, violations are stored only at the database that is responsible for correcting the constraint, *i.e.*, the participant specified by the violations->responsibility attribute. However, violations stored at any participating database are available for querying by other participants also. Storing the violations and the information derived at compile time, is useful because this information can be queried later in the design process. The next section describes how we store the derived information in repositories to assist in constraint management.

5 Constraint Repositories

Constraint repositories are used to store constraint specifications, and all the information derived at compile time. Constraint repositories serve two major functions: (1) to answer queries about the configuration constraints and (2) to assist in efficient run-time checking of constraints. Function (1) helps relate the high-level project requirements to the specific constraints that check these requirements. Often these high-level requirements are passed by the client as design criteria documents to the project design team. These criteria documents in turn refer to specific codes and regulations such as Uniform Building Code (UBC) and American National Standards Institute (ANSI) regulations. The configuration constraints used to check these regulations can be tied to the requirements in the criteria documents to assist in requirement tracking for a configuration [Tiw93]. Similar ideas for processing design standards were presented in [GF87, YL93]. The repositories assist in answering queries like:

- What electrical constraints were violated during the conceptual design phase of configuration?
- Which constraints satisfy the ANSI requirements for the HVAC equipments in the mechanical room?
- Which objects violate constraint C_5 at 30% design review?

Two kinds of repositories need to be maintained: local repositories for the discipline-specific information and a global repository for the interdisciplinary information. Both kinds of repositories can be implemented in a distributed manner using the local discipline-specific databases. We refer to the database objects used to maintain the repositories as *catalogs*. In the next subsection, we will present the details of constraint catalogs which support queries on constraints, store violations, and are used by the constraint validation algorithms presented in appendix B. Catalogs are maintained both at the global and local databases in order to speed up the distributed constraint checking process [GT93].

5.1 Catalogs

Two types of constraint catalogs are built at constraint compile time – global and local constraint catalogs. The global catalog is part of the configuration database, and stores the global information about the configuration constraints, that is relevant to all the participants. By contrast, local catalogs contain information relevant to a particular participant. We will briefly explain the contents of these catalogs, and give partial instances of each using our running example 2.1.

Global Catalog

Each constraint has an entry in the global catalog. The global constraint fragment for a constraint constitutes this entry. As described in Section 2, the GCF contains an identifier for the constraint, the constraint specification, different databases referred, the global query for checking the constraint, the query that computes the contents of the notification, *etc.* There is also administrative information like, the creator of constraint, responsible participant for its violation, the time of creation, authorization, and privileges. A constraint can be a part of many different persistent and what-if configurations. Each configuration specifies which constraints are part of it (the constraint does not have any information about the configurations that it is part of). This information can then be updated without affecting the constraint catalogs.

Global Constraint Catalog					
Constr -Id	Constraint Specification	Databases Referred	Global Query to compute Conflict Set	Notification	...
C_g	Designer::Columns.Weight $\geq$ all (**select** Cranes.Capacity **from** Contractor::Cranes **where** Cranes.Floor_Id = Columns.Floor_Id)	Designer Contractor	**select** Column_Id from Columns **where** Weight $\geq$ all (**select** Crane_Capacity **from** Contractor::Cranes **where** Cranes.Floor_Id = Columns.Floor_Id)	Notify: Designer, Contractor, Project Manager (results of the global query)	...
⋮	⋮	⋮	⋮	⋮	

Figure 5: Global Catalog entries for constraint C_g from Example 2.1

Figure 5 illustrates part of the global catalog for our running Example 2.1. Note, the query for checking the constraint is derived at constraint compile time and stored in a ready-to-use format. At run-time, when a constraint is being checked, the query can simply be fetched and executed. Similarly, the notification entry specifies the contents of the notification to be sent to the participants in case of a violation.

Local Catalogs

Local catalogs are maintained at different discipline-specific databases and store the constraint fragments, invalidating operations, local tests, and other information derived at compile time, that is relevant to the specific database. Bringing the constraint information closer to the source of changes is an optimization decision. The local catalogs contain the local constraint fragments as described

Local Constraint Catalog @ Designer			
Constr-Id	Monitors	Local Checks (LC)	If LC fails
C_g	insert into Columns	**select** C.Id **from** Columns C, Inserted-Columns IC **where** C.Floor_Id = IC.Floor_Id **and** C.Weight $\geq$ IC.Weight	**select** * **from** Inserted-Columns
C_g	update Columns	**select** C.Id **from** Columns C, Updated-Columns UC **where** C.Floor_Id = UC.Floor_Id **and** C.Weight $\geq$ UC.Weight	**select** * **from** Updated-Columns
⋮	⋮	⋮	⋮

Figure 6: Local catalog entries at the Designer's database for constraint C_g

in Section 2. For instance, the local catalogs contain an entry for each invalidating operation which needs to be monitored at that database. Particular database systems may provide triggers that provide this monitoring [WF90]. 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 primary key or attribute-range restriction checks [Ull89]. For instance, if the floor_id of the columns is non-updatable (the columns cannot be moved from one floor to the other) then there is no need to monitor such an operation and it can be removed from the list of invalidating operations in Example 2.2. Such attribute-level checks can be enforced by the local DBMS that stores information about the column objects.

Figure 6 contains the local catalog for our running example 2.1. The local catalog also stores the local tests. When an invalidating operation occurs, the corresponding local test is executed in an attempt to first check the constraint locally. If such a check succeeds, we are guaranteed that the constraint is not violated because of that invalidating operation. If the local check fails the alternative query ("If LC Fails" query in Figure 6) is executed against the database and its result are used to initiate a global check. Therefore, the global test is initiated only for newly inserted/deleted/updated objects and not for the entire database. Local check entries are derived at compile-time as mentioned in Section 3.2.

6 Related Work

In [Hol82] Holtz described symbolic algebra and dependency driven expression to manipulate design constraint for consistency management in design. He suggested using decision tables to express constraints imposed by building design specifications, such as codes, design standards and other normative documents. Similar to our work, Holtz uses the actual values of the design attributes for checking constraints. However, his work did not address where the design data and constraints were stored and how they would be accessed. El-Bibany [EB91] worked on a framework for coordination in design-construct projects. The coordination system ConCoord, uses a centralized database to store project data while AI planning and simulation techniques are used to achieve coordination between project participants. In [RF86], Rasdorf and Fenves emphasize the importance of automated representation and processing of design constraints in databases. Their work is applicable to a centralized database, and they proposed a mechanism of *augmented relations* where additional status attributes were appended to the database relations. This approach can serve as a monitoring facility to automate the loading of design data in engineering databases.

Garret and Fenves [GF87] presented a knowledge independent design strategy that used information about the design standards independent of the design process. Standards requirements are used to formulate the set of constraints that need to be satisfied. However, the emphasis of the study was on selection of applicable set of requirements for different design situations while we treat the requirements as input and focus on their mapping to distributed database objects and constraint checking on them. Yabuki and Law presented an Object-Logic model for representation of design codes and performing conformance checking. Their approach deals with single object constraints and did not address the issue of distributed data sources. The DICE project [SAL92] addresses the issues of engineering coordination and communication through a centralized blackboard and a global 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 information management in similar environments.The KADBASE project [HR85, How91] sought to provide a transparent interface to knowledge-based applications and databases in a heterogeneous and distributed environment. They maintain a global data dictionary that allows applications to query the databases without having to worry about data distribution and data representation because that information is stored in the global dictionary. KADBASE's idea of a global data dictionary has been effectively extended to global constraint repositories in our framework. The idea of invoking, activating and deactivating constraints presented in [FR85] are very relevant in our framework.

In [TH93], we presented an architecture of the constraint management system for collaborative AEC databases and described a constraint language for expressing inter-disciplinary constraints on these databases. We also incorporated a notification scheme to notify participants about the design conflicts discovered through constraint violations. The framework in [TH93] can be easily extended to support constraint checking on configurations. [Kri93] discusses how to extend this framework to support constraint checking on design versions at individual databases. The techniques for local validation of constraints presented in [GW93] are also directly applicable. [Tiw93] explored the application of automated constraint checking for ongoing space-launch facility and power plant projects.

7 Conclusions

Project configurations involve multiple databases and a large number of constraints specified on the databases. The constraints capture the high-level functional requirements specified by the client and industry standards and regulations. These requirements need to be satisfied as the design develops through the project, for example from the conceptual design stage to the detailed design and construction. In this paper, we presented a configurations model that can support collaboration among the designers in large AEC projects. We presented constraint specification and checking techniques that can be used to check inter-disciplinary constraints spanning different participants' designs. The constraints are treated separately from the design data thereby making the two issues easier to deal with and allowing constraints to be reused in other projects. In addition, independently created databases can participate in a configuration since constraints can be checked on pre-existing databases.

We presented incremental constraint checking strategies and optimizations that improve upon the brute force technique of checking all constraints against the design configurations. However, we believe that incremental constraint checking, both local tests and incremental global tests, is not very useful in the case of large changes to the configurations. In these cases, the overhead of determining when the optimizations can be applied offsets the performance gains of the optimizations. For instance, a brute force constraint checking strategy can be run overnight to baseline a persistent configurations after major design changes. Nevertheless, incremental checking and the associated optimizations will be quite useful for evaluating constraints on what-if configurations.

Constraint repositories provide the participants a facility to query the constraints defined on the configuration. These repositories can be reused from project to project especially for projects where the number of constraints can be very large, such as power plants. Repositories are also used to facilitate constraint checking and for storing violations. Keeping track of violations allows partially inconsistent configurations which are very important in realistic design scenarios where all the information may not be available or consistent. Violations can be used to help participants identify and correct design inconsistencies in a configuration. Violations also serve as a means to quantify the quality of design for example, a configuration with more than 10% of its constraints violated may be unacceptable. There is ongoing work to generalize the configuration model to support versioning schemes for the configuration components.

Acknowledgements

Most of the ideas presented in this paper were a result of discussions with members of the Collaborative Environment for Design of Buildings (CEDB) group at Stanford. Professors Craig Howard, Kincho Law, Paul Teicholz, Jeffrey Ullman, Dr. Arthur Keller, and Karthik Krishnamurthy provided valuable feedback on all the ideas. Professors Craig Howard, and Jennifer Widom gave us insightful comments on the manuscript. Funding for the current project is provided by NSF grant IR-9116646.

References

[CG85] S. Ceri and G. Gottlob. Translating SQL into Relational Algebra: Optimization, Semantics and Equivalence of SQL Queries. *IEEE Transaction of Software Engineering*, 11(4):324–345, April 1985.

[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.

[EB91] Hossam El-Bibany. *ConCord: A constraint based framework for coordination in a generic collaborative Design-Construct Environment.* PhD thesis, Stanford University, Department of Civil Engineering, 1991.

[FR85] Steven J. Fenves and William J. Rasdorf. Treatment of engineering design constraints in a relational data base. *Engineering with Computers*, 1:27–37, 1985.

[GF87] J. H. Garrett and Steven J. Fenves. A Knowledge-Based Standards Processor for Structural Component Design. *Engineering with Computers*, 2(4):219–238, 1987.

[GT93] Ashish Gupta and Sanjai Tiwari. Distributed Constraint Management for Collaborative Engineering Databases. In *Second International Conference on Information and Knowledge Management, Washington, D.C.* ACM, 1993.

[GW93] Ashish Gupta and J. Widom. Local Verification of Global Integrity Constraints in Distributed Databases. In *ACM SIGMOD International Conference on Management of Data*, pages 49–59, 1993.

[Hol82] N. M. Holtz. Symbolic Manipulation of Design Constraints - An Aid to Consistency Management. Technical Report DRC-02-012-82, Design Research Center, Carnegie Mellon University, 1982.

[How91] H. C. Howard. Linking design data with knowledge-based construction systems. Technical Report CIFE Spring Symposium Proceedings, Center for integrated Facility Engineering, Stanford University, 1991.

[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.

[Kri93] Karthik Krishnamurthy. Version and Configuration Management for Collaborative Design. *An Unpublished Thesis Proposal submitted to the Deaprtment of Civil Engineering, Stanford University*, 1993.

[RF86] William J. Rasdorf and Steven J. Fenves. Constraint enforcement in a structural design data. *Journal of Structural Engineering*, 112(12):2565–2577, 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. Distributed AEC Databases for Collaborative Design. In *To Appear in The Journal of Engineering with Computers*. Springer-Verlag, 1993.

[THL93] Sanjai Tiwari, H. C. Howard, and Raymond E. Levitt. A Design Notification Scheme for Distributed AEC framework. In *The Management of Information Technology for Construction – First International Conference, Singapore*. ASCE, August 1993.

[Tiw93] Sanjai Tiwari. Information Management in Space Launch Facility Projects. *An Unpublished Research Report submitted to the Research & Development Group, Bechtel Corporation, San Francisco*, September, 1993.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1 and 2. Computer Science Press, Rockville, Maryland, 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.

[YL93] N. Yabuki and K. Law. An Object-Logic Model for the Representation and Processing of Design Standards. *Engineering with Computers*, 9:133–159, 1993.

A Example Constraint Specifications

Below we present example constraint specifications from a Power Plant project. The project involved many disciplines, such as architectural, mechanical, electrical, structural, but to name a few. The source discipline of a constraint and the discipline responsible for handling its violations are indicated. Each constraint specification is followed by a query that computes the contents of the violation to be notified.

1. *Check that all the door frames for the personnel rooms in the architectural database are made of fiberglass.*
   Constraint->Source: Architect Violation->Responsibility:Architect

```
Architect::Frames :
        exists (Select   Doors.Id
                From     Architect:: Doors
                Where    Doors.Frame-Id = Frames.Id
                  and    Doors.Space = "Personnel"
                  and    Frames.Material != "fiberglass");
        Notify(Architect) (Frames.Id, Frames.Material);
```

   Query to compute Violations

```
Select   Frames.Id, Frames.Material
From     Architect::Frames
Where    exists (Select   Doors.Id
                 From     Architect::Doors
                 Where    Doors.Frame-Id = Frames.id
                   and    Doors.Space = "Personnel"
                   and    Frames.material != "fiberglass");
```

2. *Check that all the HVAC equipments in the mechanical database are connected to the electrical grounds in the electrical database.*
   Constraint->Source: Electrical Violation->Responsibility: Electrical

```
Mechanical::Equipments :
        not exists ( Select   Grounds.Id
                     From     Electrical:: Grounds
                     Where    Equipments.Type = "HVAC"
                       and    Equipments.Grounds-Id = Grounds.Id);
        Notify(Electrical, Mechanical) (Equipments.Id);
```

   Query to compute Violations

```
Select   Equipments.Id
From     Mechanical::Equipments
```

```
Where not exists (Select  Grounds.Id
                  From    Electrical::Grounds
                  Where   Equipments.Type = "HVAC"
                    and   Equipments.Ground-Id = Grounds.Id);
```

3. *Check that there are no conduits below 7'2" in the mechanical database.*
   Constraint->Source: Client   Violation->Responsibility: Mechanical

```
Mechanical::Conduits :
        7.17 ≤ Conduits.Height
        Notify(Mechanical) (Conduits.Id, Conduits.Height);
```

   <u>Query to compute Violations</u>

```
Select  Conduits.Id, Conduits.Height
From    Mechanical::Conduits
Where   7.17 ≤ Height
```

4. *Check that the sum of the cross-sectional area of the power cables in the electrical database is less than the area of the conduit that contains them in the mechanical database.*
   Constraint->Source: Mechanical   Violation->Responsibility: Mechanical

```
Mechanical::Conduits :
        Conduits.Area ≤ ( Select  sum (Cables.Area)
                          From    Electrical::Cables
                          Where   Cables.Type = "Power"
                            and   Cables.Conduit-Id = Conduits.Id);
        Notify(Mechanical) (Conduits.Id, Conduits.Area, sum(Cables.Area));
```

   <u>Query to compute Violations</u>

```
Select  Conduits.Id, Conduits.Area, sum(Cables.Area)
From    Mechanical::Conduits, Electrical::Cables
Where   Cables.Type = "Power"
  and   Cables.Conduit-Id = Conduits.Id
  and   Conduits.Id in (Select  Conduits.Id
                        From    Mechanical::Conduits
                        Where   Area ≤ (Select  sum (Cables.Area)
                                        From    Electrical::Cables
                                        Where   Cables.Type = "Power"
                                          and   Cables.Conduit-Id = Conduits.Id));
```

5. *Check that there are no columns within 15' of the centerline of the turbine at floor levels 1-3 in the in the structural database.*
   Constraint->Source: Client   Violation->Responsibility: Structural

```
Client::Turbines :
      exists ( Select  Columns.Id
               From  Structural:: Columns
               Where Columns.Floor-Level ≥ 1
                 and   Columns.Floor-Level ≤ 3
                 and   Compute-Distance(Columns.Location, Turbines.Center-Line)≤ 15);
      Notify(Structural) (Columns.Id, Columns.Floor-Level)
```

Query to compute Violations

```
Select   Columns.Id, Columns.Floor-Level
From     Client::Turbines, Structural::Columns
Where    Columns.Floor-Level ≥ 1
   and   Columns.Floor-Level ≤ 3
   and   Compute-Distance(Columns.Location, Turbines.Center-Line)≤ 15;
```

6. *Check that the ratings of all Commercial-off-the-shelf (COTS) equipment meet the specifications in the electrical database as per their rated capacity .*
   Constraint->Source: Project Manager   Violation->Responsibility: Electrical

```
Electrical::COTS :
      COTS.EMI-Rating not in ( Select  Ratings
                               From  Electrical:: EMI-Specs
                               Where COTS.Type = "Electrical"
                                 and   COTS.Capacity = EMI-Specs.Capacity);
      Notify(Electrical) (COTS.Id, COTS.EMI-Rating, EMI-Specs.Rating)
```

Query to compute Violations

```
Select   COTS.id, COTS.EMI-Rating, EMI-Specs.Capacity
From     Electrical::COTS, Electrical::EMI-Specs
Where    COTS.Capacity = EMI-Specs.Capacity
   and   COTS.Id in (Select  COTS.Id
                     From    Electrical::COTS
                     Where   COTS.EMI-Rating not in
                             (Select  Rating
                              From    Electrical::EMI-Specs
                              Where   COTS.Type = "Electrical"
                                and   COTS.Capacity = EMI-Specs.Capacity))
```

B Incremental Constraint Checking Algorithms

This section describes the algorithms for checking constraints when a configuration changes. There can be two kinds of changes to a configuration: (1) to the constraints $\Delta\mathcal{C}$ and (2) to the participating databases $\Delta\mathcal{D}$. We describe how constraint checking proceeds in each of these two cases.

Change to Constraint(s)

In case constraints change, the global constraint fragment (GCF) stored in the global catalog is executed. No local optimization can be done in this case because each new constraint *has* to be evaluated against all the databases it refers to. We refer to the change to constraints as $\Delta\mathcal{C}$.

Algorithm B.1
Input: $\Delta\mathcal{C}$ *% Contains the new inserted constraints and the old deleted constraints*
Method
  for each newly inserted constraint $C \in \Delta\mathcal{C}$
    Retrieve the global test query from global catalog.
    Execute global test query against the database to compute *conflict_set*.
      % conflict_set refers to the set of violations
    Add the *conflict_set* to the set of violations $\mathcal{V}$.
  for each deleted constraint
    Delete the associated violations from $\mathcal{V}$.
$\diamond$

Change to Database(s)

We refer to the change to the databases as $\Delta\mathcal{D}$.

Algorithm B.2
Input: $\Delta\mathcal{D}$ *% $\Delta\mathcal{D}$ contains $<O, op>$ pairs where op is the operation on object O*
Method
  for each object, invalidating operation pair $<O, op>$ in $\Delta\mathcal{D}$
    for each corresponding monitor entry M from the local catalog
      Execute the local test in M against the local database
      if the local test fails
        Retrieve "If LC fails" query from M. *% computes information to be used for global check*
        Execute "If LC fails" query to compute *local_conflict_set*.
          % local_conflict_set *contains the data to be used in the global test*
        Retrieve global test query for constraint C corresponding to $<O, op>$.
        Instantiate global test query using objects in *local_conflict_set*.
        Execute instantiated global test query to compute *conflict_set*
          % conflict_set refers to the set of violations
    Add the *conflict_set* to the set of violations $\mathcal{V}$.
$\diamond$

Note: Algorithm B.2 ignores the violations that $\Delta\mathcal{D}$ could correct.