Carnegie Mellon University
Research Showcase @ CMU

Department of Civil and Environmental Engineering

Carnegie Institute of Technology

1982

Treatment of engineering design constraints in a relational database

Steven J(Steven Joseph) Fenves
Carnegie Mellon University

William John. Rasdorf

Follow this and additional works at: http://repository.cmu.edu/cee

Published In

.

This Technical Report is brought to you for free and open access by the Carnegie Institute of Technology at Research Showcase @ CMU. It has been accepted for inclusion in Department of Civil and Environmental Engineering by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

TREATMENT OF ENGINEERING DESIGN CONSTRAINTS IN A RELATIONAL DATABASE

by

S,J. Ferwes & W.J. Rasdorf

December, 19<'2

DRC-12~1'4~32

TREATMENT OF ENGINEERING DESIGN CONSTRAINTS IN A RELATIONAL DATABASE

Steven J. Fenves[1] and William J. Rasdorf[2]

1. INTRODUCTION

There is considerable interest and activity in the development of engineering design databases. Such databases are viewed as the primary integration mechanism between the various design processes. Developers of such databases are naturally interested in database management systems (DBMS), and are exploring ways in which existing DBMS's can be adopted or extended to engineering design applications. There is particular interest in relational databases, because of their formal basis and flexibility.

A major aspect of engineering design involves the evaluation and satisfaction of constraints. Constraints arise from many sources: they may represent functional relations (voltage equals current times resistance), they may be externally imposed by codes and standards governing acceptability, or they may represent design objectives or the designer's particular design "style." The ability to represent and process a wide variety of such constraints is a necessary ingredient of any engineering design database. This is especially true in databases integrating several design processes, where the DBMS must serve not as a passive repository of data, but as an active agent performing many of the consistency and integrity checks that are currently done manually.

In this paper, we propose a mechanism for representing and processing design constraints. The mechanism can be used for checking constraints, i.e., determining whether they are satisfied or violated, as well as for assigning attribute values such that the applicable constraints are thereby satisfied Furthermore, the mechanism provides flexibility in sequencing the enforcement of constraints, by allowing new constraints to be applied to a pre-existing state of the database as well as to all subsequent transactions on the database. In both of these respects, the mechanism proposed appears to have applications beyond engineering design.

---

[1] University Professor of Civil Engineering, Carnegie-Melton University. Pittsburgh, PA 15213

[2] Assistant Professor of Civil Engineering. North Carolina State University. Raleigh NC 27650. Formerly Research Assistant Department of Civil Engineering. Carnegie-Mellon University. Pittsburgh. PA 15213.

UNIVERSITY LIBRARIES
CARNEGIE-MELLGN UftI^ERSiTY
PITTSBURGH. PENNSYLVANIA 15213

2. THE RELATIONAL MODEL

This section presents a brief overview of the salient features of the relational model, to serve as an introduction to the incorporation of constraints in the model.

2.1. THE STRUCTURE OF RELATIONS

A relational model is a single level model consisting of a collection of relations represented in two dimensional tabular form [7]. The rows of a relation are called tuples and its columns are called attributes. Attribute values are drawn from an underlying domain which represents all legal uses of instances of the domain. Each tuple represents an entity and contains an attribute value from each domain. All tuples are distinct; duplicates are not permitted. Tuples and attributes have no order; they may be arbitrarily interchanged without changing the data content and meaning of the relation. Tuples are accessed by means of a key, a single attribute or a combination of attributes that uniquely identifies one tuple.

2.2. REPRESENTATION OF RELATIONS

Throughout this paper a standard shorthand notation will be used to represent relations. The form of the notation is as follows:

RELATIONname (ATTRIBUTE1name, ATTRIBUTE2name, . . .).

To use a specific example, a database supporting architectural space planning may contain the relation:

ROOMS (<u>roomID</u>, area, breadth, width).

The name of the relation is listed first followed in parentheses by the names of all of its attributes. The underlined attribute of a relation is the key.

2.3. FUNCTIONAL DEPENDENCE AND NORMALIZATION

Normalization is the process of removing intrarelation dependencies among attributes of relations. The ROOMS relation introduced above can be used to illustrate functional dependence. The roomID attribute is the key of the relation. For each roomID, there is precisely one corresponding area, breadth, and width value. Each of the attributes area, breadth, and width are therefore functionally dependent on, and functionally determined by, the roomID.

The ROOMS relation also contains a transitive dependency. A transitive dependency is a relationship between non-key attributes, i.e., there exists an attribute that depends on one or more attributes other than the key. In the ROOMS relation, a room's area is dependent not only on the roomID but on the remaining non-key attributes as well, since area = breadth * width. This dependency could be removed by normalization and the ROOMS relation replaced by two new relations as follows:

ROOMA (roomID, area)
ROOMBW (area, breadth, width).

It is to be noted that this form of normalization assumes that there is only one particular combination of breadth and width which yields a given room area, which may not correspond to the designer's intent

2.4. INTEGRITY CONSTRAINTS

All relational DBMS's support single attribute integrity constraints which delimit the domain of legal attribute values. For example, in the ROOMS relation one may wish to impose

RULE area IN rooms > 0.
RULE breadth IN rooms > 0.
RULE width IN rooms > 0.

In [2], Fagin suggests that a broader class of integrity constraints which should be enforced by a relational DBMS is the single-tuple constraint

$$f(t) \tag{1}$$

where *f* is a constraint and t is an arbitrary tuple of relation R. A tuple t is said to satisfy constraint *f,* and *f* is said to hold for tuple t, if ^(t) is true [2]. Otherwise t does not satisfy *f.* The purpose of this paper is to formalize this class and suggest mechanisms for their enforcement

Extensions of integrity constraints referring to the semantic integrity of the data have been widely discussed in the DBMS literature [8], [1]. The constraints dealt with in this paper may be viewed as a special form of semantic integrity constraints, where the semantics or meaning of the data relate to the requirements of engineering design.

3. DESIGN CONSTRAINTS

Engineering design deals with the evaluation and satisfaction of many constraints. A design is considered satisfactory if it satisfies all applicable constraints.

In this paper, we restrict our attention to single-tuple constraints of the form given in (1).

3.1. CANONIC FORMS OF CONSTRAINTS

Using the terminology of mathematical optimization, the two canonic forms of constraints *are:*

$$h(a_i) = 0, \text{ and} \tag{2}$$

$$g(a_i) \leq 0 \tag{3}$$

where the a_i are attributes of a relation R (i = 1 ... k; k = arity of R) and h and g represent respectively, equality and inequality functional dependencies among the attributes a_i.

These constraints can be illustrated using the ROOMS relation introduced above. The equality constraint discussed above is:

Narea, breadth, width) = area - breadth « width = 0. (4)

The designer's preference or "style" may dictate the following inequality constraints on the •aspect ratio of the rooms:

g_1(breadth, width) = breadth - 2 * width £ 0. (5)

g_2(breadth, width) = width - 2 « breadth £ 0. (6)

These two canonic forms of constraints will be referred to as checking constraints. They simply check the relationship between the attributes of the expression. Their evaluation results in a boolean value of true or false, which can be interpreted as satisfied or violated, respectively. To perform the evaluation, values must be known for all of the attributes in the constraint

3.2. ASSIGNMENT USING CONSTRAINTS

For design purposes, it is frequently necessary to recast or paraphrase constraints into an assignment expression:

$$a_j := h(a_i) \quad (7)$$

where ":=" is the assignment operator.

We assume that the original constraint $h(a_i) = 0$ can be explicitly solved for any a_j, that is, i # j on the right-hand side (RHS) of expression (7). Thus the expression allows for the assignment of a value to one attribute as a function of the values of the remaining attributes consistent with the constraint Implicit expressions, which include a_j on the RHS, are not considered in this paper.

3.3. EXTENDED FUNCTIONAL DEPENDENCE

It can be seen from equation (7) that a_j is functionally dependent on the arguments a_i, i # j, with respect to the function h. Furthermore, since each of the attributes a_i in the original constraint (2) can be similarly expressed as a function of the remaining attributes, we say that the attributes a_i are fully functionally interdependent with respect to the constraint $h(a_i) = 0$. Similar dependencies occur with the inequality constraint $g(a_i)$ £ 0, which can also be paraphrased as an assignment expression.

3.4. INGREDIENTS AND DEPENDENTS

Assignment expressions for any attribute a_j formed from an equality or inequality constraint provide a dynamic use of the functional dependencies among the attributes. Instead of checking the conformance of all existing attribute values as constraints (2) and (3) suggest assignments of new attribute values can be made in accordance with expression (7). Distinctions can then be made between dependent and ingredient attributes.

Ingredient attributes are all of the attributes required to evaluate an assignment expressioa The dependent attribute is the single attribute expressed as a function of the remaining attributes. In expression (7), a_j is the dependent attribute and the a_i are the ingredient attributes.

Rewriting constraint (4) in the form of one of its possible assignment expressions,

area := breadth * width,

area is the dependent attribute of the expression and breadth and width are the ingredient attributes.

4. STANDARD TREATMENT OF CONSTRAINTS

It is instructive to review the extent to which the standard treatment of intra-relation dependencies, namely normalization, may be used for enforcing design constraints.

4.1. POTENTIAL NORMALIZATION

The distinction between the two canonic forms of constraints is arbitrary, and will be removed in the following. This is especially true if the attributes in the h-form are defined over real domains, in which case the constraint is normally given in the form:

$$g(a_i) = (|h(a_i)| - e) \pounds 0 \qquad (8)$$

where | | denotes the absolute value and * is some specified tolerance.

The reason for introducing the h-form is that conceptually the h-form constraint could be enforced through normalization. Normalization removes from a relation all attributes that are not fully functionally dependent on the key.

For any attribute a_j selected from (7), normalization can be achieved by simply projecting that attribute out of the relation. The dependent or "redundant' attribute a_j can then be obtained from (7) using the current values of the ingredient or "independent" attributes a_i (i * j). Alternatively, the original relation can be split into two normalized relations, as illustrated in Section 2.3. However, as noted there, that approach introduces another assumption or constraint

The objection to normalization in design applications is that it is frequently problematic which attribute is dependent or independent Furthermore, as design proceeds, attributes change from ingredient to dependent and vice versa Normalization therefore restricts the free flow of the design process.

Continuing with the space planning example, in one situation the designer may first manipulate room areas until some overall constraint on space is tentatively satisfied He may then proceed to lay out dimensions, adjusting room areas only as necessary. In another scenario, the designer may start with dimensions derived from a fixed grid and then determine the room areas. Thus, at various times, he may wish to use any one of the three possible assignment expressions paraphrased from constraint (4):

$$\begin{aligned} \text{area} &:= \text{breadth} * \text{width} \\ \text{width} &:= \text{area} / \text{breadth} \\ \text{breadth} &:= \text{area} / \text{width} \end{aligned} \tag{9}$$

Clearly, normalization based on an a-priori selection of a dependent attribute is not a flexible enough mechanism to satisfy these needs. A new mechanism is needed which should also handle the g-form of functional dependency, which cannot be removed by normalization

4.2. DIRECT ENFORCEMENT OF CONSTRAINTS

A potential solution is to incorporate constraints (2) and (3) directly into the integrity rules enforced by the database, so that transactions (insertions and updates) on tuples satisfying the constraints are accepted while transactions violating them are rejected This mechanism is a direct extension of the single-attribute integrity rules found in most relational DBMS's, and is essentially the approach taken in enforcing semantic integrity. However, the integrity rules in current relational DBMS's are defined on the schema, and are active as soon as a relation is instantiated

This mechanism falls short in the flexibility required, in two aspects. First in the early phases of design, when information is gathered from many sources and tentative solutions are explored, it is not realistic (or even feasible) to require that all constraints pertaining to the completed design be immediately satisfied on the first trial design Nor is it realistic to assume that all of the data needed to evaluate the constraints will be available. Constraints will automatically be violated if uninitialized values exist for attributes of a tuple. What is needed is a way to record, control, and change the status of each constraint as the design proceeds. Second, this mechanism does not support the assignment of attribute values subject to the applicable constraints.

5. AUGMENTED RELATIONS

A new type of relation, referred to as an augmented relation is proposed herein.

5.1. REPRESENTATION OF CONSTRAINTS

The original relation schema is augmented with additional attributes recording the status of the constraints, one for each constraint on the relation, defined as follows:

$$a_k := (h_k(a_i) = 0) \tag{10}$$

or

$$a_k := (g_k(a_i) \text{ £ } 0). \tag{11}$$

where the expression on the RHS represent the constraints and the a_k are the new attributes, one for each constraint k.

The additional constraint attributes are referred to as status attributes. The domains of the constraint status attributes are boolean, with the possible values of true and false, interpreted as satisfied and violated, respectively. Thus, for the example previously used, the ROOMS relation is modified to:

ROOMS2 (roomID, area, breadth, width, areaOK, shapeOK)

where the two new constraints monitored by the status attributes are defined by the expressions:

```
areaOK : = abslarea - breadth * width) <= 0.01
shapeOK := (breadth <= 2 * width) AND (width <= 2 * breadth).        (12)
```

These expressions can be coded as boolean constraint functions and associated with the corresponding status attribute. The first constraint expression would be coded in Pascal as:

```
FUNCTION checkarea (area, breadth, width : real) : boolean;
  BEGIN
     checkarea := abslarea - breadth * width) <= 0.01
  END.
```

The function (or, preferably, the linkage between the DBMS and the function) must automatically set the value of the constraint status attribute to false if any one of its arguments (values of ingredient attributes in the defining expression) is undefined, i.e., has the special value of uninitialized. In this manner the proposed mechanism can always directly evaluate the status of the constraint regardless of the availability of data

We emphasize that each status attribute is fully functionally dependent on the ingredient attributes that are arguments to its constraint function. Thus, instead of enforcing integrity by normalizing out an arbitrary dependent attribute, we monitor the constraint status by introducing an additional functional dependency.

5.2. ENFORCEMENT OF CONSTRAINTS ON AUGMENTED RELATIONS

The enforcement mechanism is predicated on the typical design sequence of initially selecting trial values for attributes, then successively "firming up" the design by enforcing conformance with a succession of key constraints until eventually the finished design satisfies all constraints. The mechanism also accommodates the frequent situation where a candidate design turns out to have undesirable properties or to be infeasible with respect to some constraints, in which case constraints need to be withdrawn, new trial attribute values selected, and another design iteration initiated

The proposed enforcement mechanism uses three new DBMS commands. Initially, none of the constraints are active, i.e., all constraint status attributes have their initial value set to false (or violated). The command to apply a constraint to the current state of the database is of the form:

INVOKE Constraint function> ON <relation>

where <constraint function> is the name of the constraint function to be applied Multiple constraints can be invoked by a single command This command causes a batch process to be performed, applying the constraint function to each tuple in the relation in turn, and recording the resulting value (true or false) of the constraint status attribute. For the sample function shown, the command:

INVOKE checkarea ON rooms2

causes the assignment

areaOK := checkarea (area, breadth, width)

to be performed for each tuple.

As part of the process, the non-conforming tuples are located and output This operation is equivalent to a standard query, e.g., for the sample relation R00MS2:

SELECT FROM rooms2 WHERE (areaOK # true).

Remedial action can then be taken to bring these tuples into conformance.

A relation holds design information about many objects, with each tuple representing one. Because the nature of the design process is such that objects may be designed at varying times and in varying orders it is not always realistic to impose a constraint invocation on the entire relation. Therefore, the INVOKE command can be combined with a SELECT FROM clause, allowing invocation of the constraint only on those tuples satisfying the selection criterion.

The command to apply a constraint on all subsequent transactions is of the form:

ACTIVATE Constraint function> ON <relation>.

The effect of ACTIVATE is equivalent in outcome to a standard integrity rule of the form:

RULE Constraint status attribute> IN <relation> # false.

That is, for each transaction (insertion or update) on a tuple, the function for evaluating the constraint status attribute is invoked for that tuple only and the transaction is accepted if the constraint status attribute evaluates to true or rejected if it evaluates to false.

Conceptually, there is no need to INVOKE a constraint function on the current state of the database before that constraint function is ACTiVATEd for subsequent transactions. However, there is no assurance that every tuple will be involved in a transaction. Therefore, complete integrity of the relation can be assured only if ACTIVATE is preceded by an INVOKE This can be achieved by having ACTIVATE automatically initiate the corresponding INVOKE and proceed only if all tuples conform

Finally, a command of the form:

DEACTIVATE Constraint function> ON <relation>

suspends the enforcement process and the relation reverts to its initial mode with respect to the specified constraint(s), allowing modifications to be made without checking The constraint can be subsequently re-INVOKEd and re-ACTIVATEd.

When constraints can be defined at any time during the life of a database, it must be possible to apply a newly defined constraint on the attributes of a relation on a tuple by tuple basis, not only to all subsequent transactions but also to the pre-existing state of the relation, i.e., to all previous transactions. The three commands described above satisfy these requirements. The INVOKE command brings the status attributes up to date from all previous transactions, while ACTIVATE command insures constraint compliance for all subsequent transactions. The commands also enable the integrity of a relation with respect to a constraint to be restored after the use of that constraint has been suspended by deactivation.

5.3. ASSIGNMENT OF ATTRIBUTES SATISFYING EQUALITY CONSTRAINTS

Once a mechanism is implemented to invoke a constraint function (either in batch or one transaction at a time) and evaluate and store a constraint status attribute value based on the current values of its ingredient attributes, a major further step can be taken. It was pointed out in Section 4.1 that at various stages of design, all paraphrases $a_j := h(a_i)$, $i * j$ obtained from a given equality constraint $h(a_i) = 0$ may prove useful to the designer. That is, he may wish to evaluate and store any attribute value dependent on the current values of the other attributes appearing in the constraint

This extension can be readily achieved in the proposed constraint enforcement mechanism by defining a set of <u>assignment procedures</u> for each constraint. Each procedure returns values of two attributes: the selected dependent attribute evaluated from $a_j = h'(a_i)$; and the constraint status attribute. The latter is automatically assigned the value true (or satisfied), since the former is evaluated in such a way that the constraint expression is satisfied. The reason for requiring that the procedure return two results, i.e. that it be a procedure rather than a function, is that all enforcement control is performed on the constraint status attribute, as described in Section 5.2. Thus, the command:

INVOKE <assignment procedure> ON <relation>

will invoke the procedure in turn for each tuple, compute and store the value of the dependent attribute, and set the constraint status attribute to true. A subsequent SELECT FROM query to locate violated status attribute tuples would return an empty relation. Similarly once a command of the form:

ACTIVATE <assignment procedure> ON <relation>

is given, the same process is performed dynamically for each tuple in the relation affected by the transaction.

Continuing with the example, a procedure to assign the area based on the first expression in (9) would be coded in Pascal as:

```
PROCEDURE setarea (breadth, width : real;
                   Var area : real; Var areaOK : boolean);
  BEGIN
     area    := breadth * width;
     areaOK := true
  END.
```

Three comments are in order. First, as discussed in Sections 3.2 and 4.1, the attributes included in h-form equality constraints exhibit full functional interdependence so that relations such as the ROOMS relation of Section 2.2 are not in normal form, i.e., one attribute is redundant. We have suggested that this redundancy not be removed by normalization. Instead, a mechanism for both constraint enforcement and attribute value assignment is proposed. The reason for proposing the assignment procedure here is that it provides the desired flexibility for design; it also serves as an introduction to assignment procedures based on g-form inequality constraints, which cannot be normalized. Second, the presentation further emphasizes the functional dependence of the constraint status attribute, i.e., its value is known to be true when the assignment procedure is executed. The need for the "redundant" status attribute will become clear when hierarchies of constraints are introduced. Third, the sample assignment procedure shown above was "manually" derived from the original constraint expression (4). In a fully implemented

system, all such procedures could be symbolically derived using a symbolic algebra system such as MACSYMA [4].

5.4. ASSIGNMENT OF ATTRIBUTES SATISFYING INEQUALITY CONSTRAINTS

As pointed out in Section 4.1, g-form constraints cannot be normalized by conventional means. On the other hand, in defining the constraint status attribute no distinction was made between constraint expressions based on the g- or h-form The next logical step is to extend the concepts of the previous section to assignments based on inequality or g-form constraints.

The extension sought follows readily, with one fundamental but obvious distinction whereas any paraphrase $a_j = h'(a_i)$ of a constraint $h(a_i) = 0$ yields an expression for a unique value of a_j, an equivalent paraphrase a_j ^ $g'(a_i)$ of a constraint $g(a_i)$ £ 0 yields only a bound on the possible values of a_j Therefore, each g-form constraint introduces a bounded functional dependence of a_j on the remaining attributes a_i. Thus, with respect to the g-form constraint, the designer is free to choose any value for the attribute a_j subject to the bound

There are (at least) two implementation alternatives for incorporating assignment procedures based on g-form constraints in a relational DBMS. If it is intended that the designer exercise his choice within the bounds interactively, the assignment procedure based on constraints (5) and (6) may be of the form:

```
PROCEDURE set breadthfwidth : real; VAR breadth : real;
                        VAR shapeOK : boolean);
   BEGIN
      breadth := MIIM[2 * width, {user chooses}];
      shapeOK := true
   END;

PROCEDURE set width (breadth : real; VAR width : real;
                      VAR shapeOK : boolean);
   BEGIN
      width := MIN[2 * breadth, {user chooses}];
      shapeOK := true
   END.
```

In this implementation, when one of the assignment procedures is INVOKEd, the designer would be requested to choose a value of a_j for each tuple; similarly, after the procedure is ACTIVATEd, he would be asked to choose a value every time a transaction produces new values of the a_i for some tuple.

Alternately, if the designer's logic for choosing a_j is known in advance, that logic can be

directly incorporated in the assignment procedure and the resulting procedure treated in exactly the same way as for the h-form. This alternative would likely be implemented at the detailed levels of any design activity, where one typically chooses a value just satisfying a g-constraint

5.5. EXTENSION TO NON-NUMERIC CONSTRAINTS

The preceding presentation dealt with constraints on attributes with numeric domains. It should be clear that the same method can be applied to attributes whose domains are non-numeric. The only requirement is that the constraint status be stated as a boolean expression.

Extending the previous example, assume that the relation ROOMS has two additional attributes: an attribute called function with domain {'public/ 'private'} and an attribute called location with domain {'internal/ 'external'}. The constraint "a public room must have an external location" is expressed in checking form as:

```
FUNCTION usageOK (function, location : string) : boolean;
   BEGIN
    usageOK := NOT ((function = 'public') AND (location = 'internal')).
   END.
```

There are two possible assignment procedures for this constraint These are:

```
PROCEDURE setfunction (location : string, VAR function : string; VAR usageOK : boolean);
  BEGIN
     IF (location = 'internal')
          THEN function := 'private'
          ELSE function := {user chooses};
     usageOK := true
  END;

  PROCEDURE setlocation (function : string; VAR location : string;
                         VAR usageOK : boolean);
  BEGIN
     IF (function = 'public')
          THEN location := 'external'
          ELSE location := {user chooses};
     usageOK := true
  END.
```

6. MULTIPLE CONSTRAINTS

In the presentation so far, we have dealt with individual single-tuple constraints, and discussed methods for representing and checking them as well as for assigning attribute values subject to such constraints. The extension of these methods to multiple constraints (still pertaining to single tuples of single relations) involves two issues: the treatment of multiple dependents and the treatment of hierarchies of constraints.

6.1. MULTIPLE DEPENDENTS

Two constraint status attributes are said to possess shared ingredients if the intersection of the attributes in their defining constraints is not nil. That is, two constraint status attributes of the form shown in (11), say

$$a_{ki} := KM = 0) \text{ and}$$

$$a_{k2} := (h_{k2}(a_i) = 0)$$

share those ingredients a_i which appear in both h_{k1} and h_{k2}. Conversely, any such shared ingredient has both a_{kl} and a_{k2} as one of its dependents. In the space planning example, breadth and width are shared ingredients of the status attributes areaOK and shapeOK, and both of the latter are dependents of breadth and width.

As long as only constraint status attributes are evaluated using constraint functions, it is immaterial whether a given attribute a_i has multiple dependents or just one. If, however, an assignment procedure is used to assign a value to an attribute a_i possessing multiple dependents, a potential inconsistency arises. If a_i is assigned a value based on satisfying constraint h_{k1}, the status of constraint h_{k2} is potentially affected

Consistency can be maintained by including in the assignment procedure for attribute a_i additional statements to evaluate the status of all constraints dependent on a_i To illustrate, assume that the designer wishes to have available two procedures for computing the width of rooms: one based on the area equality constraint and one making the rooms square (thus satisfying the aspect ratio constraint). The following two functions and two procedures are then needed

```
FUNCTION checkarea (area, breadth, width : real) : boolean;
          BEGIN
             checkarea := abstarea - breadth * width) <= 0.0l
          END;

FUNCTION checkshape (breadth, width : real) : boolean;
          BEGIN
             checkshape := (breadth / width <= 2.0) AND
                           (breadth / width >= 0.5)
          END;
```

```
PROCEDURE setwidth1 (area, breadth : real; Var width : real;
                     Var : areaOK, shapeOK : boolean);
    BEGIN
      width    := area / breadth;
      areaOK   := true;
                  {status known, assignment based on this constraint};
      shapeOK  := checkshape (breadth, width)
                  {to get status of other dependent of width}
    END;

PROCEDURE setwidth2 (breadth : real; Var width : real;
                     Var : areaOK, shapeOK : boolean);
    BEGIN
      width    := breadth;    {designer's choice}
      shapeOK  := true;       {above satisfies shape constraint}
      areaOK   := checkarea (area, breadth, depth)
    END.
```

The functions checkarea and checkshape can be ACTIVATEd singly or together, as before. If setwidth1 is ACTIVATEd after checkshape and shapeOK evaluate to <u>false</u>, the transaction is rejected; <u>vice versa</u> for setwidth2, checkarea and areaOK.

The extension to more than two dependent constraints is straightforward We emphasize again that the functions aid procedures shown, including the evaluation of "sibling" dependent status attributes, can be symbolically generated from the original constraints.

6.2. HIERARCHIES OF CONSTRAINTS

Until now, we considered only basic constraint status attributes whose ingredients were attributes in the original, non-augmented relation. It is natural to consider additional, "higher level" constraint status attributes whose ingredients are "lower level" status attributes, forming a hierarchy of status attributes of arbitrary deptK The design process is complete when the "topmost" constraint status attribute evaluates to true for every tuple.

In the example, the "topmost" status attribute could be defined by the expression:

roomOK := areaOK AND shapeOK. (13)

To insure full consistency with respect to all "lower-level" constraints, the constraint functions corresponding to the "higher-level" status attributes must explicitly evaluate the "lower-level" constraint status attributes. Thus, the function corresponding to (13) is:

```
FUNCTION checkroom (area, breadth, width : real): boolean;
    Var areaOK, shapeOK : boolean);
    BEGIN.
      areaOK    := checkarea (area, breadth, width);
      shapeOK   := checkshape (breadth, width);
      checkroom := areaOK AND shapeOK
    END.
```

6.3. ASSIGNMENT USING MULTIPLE CONSTRAINTS

Assignment procedures based on simultaneous satisfaction of multiple constraints are possible. Holtz [3] presented a method for the symbolic manipulation of design constraints, including multiple equality and inequality constraints. In Holtz's original work, numerical values are given to some of the ingredient attributes, and bounds are generated for the remaining dependent attribute(s). It is possible to use Holtz's program with all attributes given symbolically and generate the paraphrased assignment expression for a selected dependent attribute, subject to an overall constraint

The main difference between assignments based on single and multiple constraints is that in the latter it cannot be taken for granted that the constraints can all be satisfied simultaneously. That is, the ingredient attribute values may be such that no feasible assignment can be made to the dependent attribute. Thus, assignment procedures based on multiple constraints must be written and processed so that they return either an attribute value together with overall constraint status value of true, or a status value of false.

7. CONCLUSIONS AND POSSIBLE EXTENSIONS

The objective of this paper was to present a method for handling a broad class of single-relation, single-tuple constraints typical in engineering design applications. Instead of relying on normalization - where normalization is possible at all - to remove functional dependencies, the approach presented introduces additional attributes representing the status (satisfied or violated) of each constraint thereby increasing the functional dependence of the relation.

A direct consequence of this approach is that passive constraint checking can be readily extended to active assignment of attribute values that automatically satisfy constraints. In addition, a flexible constraint enforcement mechanism permitting dynamic control is achieved

A prototype system implementing many of the components presented has been programmed in Pascal [5]. Some of these components are currently being implemented as an extension of the consistency rules of RIMS [6].

The next logical step is to extend the method to multi-tuple and multi-relation constraints. We can only offer preliminary thoughts on such an extension. The critical question is the following given a constraint evaluation function or constraint-based assignment procedure, where are their arguments (the attributes needed for the evaluation) to be found in the database? In this paper, it is assumed that all needed arguments are attribute values of the current tuple. In the most general case, the arguments may include

attribute values from other tuples of the current relation or from other relations. Using the space planning example for the last time, constraints on a room may involve attributes of:

1. the city, building, floor, etc. "owning" or containing this room;
2. the walls, equipment, people, etc. "owned" by or contained in the room; or
3. adjacent (or otherwise related) rooms.

From preliminary investigations, it appears that the method proposed can be extended to such cases, provided that the functions and procedures themselves have access to the DBMS and can use PROJECT, SELECT, and JOIN, as well as aggregate operations on relations (e.g., to evaluate constraints such as "room area ^ sum of areas of equipment in room"). The issues of implementation efficiency are the same as those discussed in [1].

Acknowledgements

Portions of this work were sponsored by the National Science Foundation under grant MCS7822328, entitled "Data Base Methods for Design" We wish to acknowledge the helpful suggestions of Dr. Daniel R Rehak, Assistant Professor of Civil Engineering, and Mr. Kangcher Shen, Visiting Scholar from Chongquing Institute of Communications, Peoples' Republic of China

REFERENCES

[1] Bernstein, P., Blausteia B. and Clarke, EM.
Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data
Proceedings of the Sixth International Conference on Very Large Databases , October 1-3, 1980.
IEEE

[2] Fagin, R.
A Normal Form for Relational Databases That is Based on Domains and Keys.
ACM Transactions on Database Systems 6(3):387-415, September, 1981.

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

[4] Mathlab Group.
MACSYMA Reference Manual, Version Nine
MIT Laboratory for Computer Science, Cambridge, Mass., 1977.

[5] Rasdorf, W. J.
Structure and Integrity of a Structural Engineering Design Database.
Technical Report DRC-02-14-82, Design Research Center, Carnegie-Mellon University, Pittsburgh, PA, April, 1982.

[6] Erickson, W. J., Gray, F. P., Limbach, G
Relational Information Management System
Version 5.0 edition, Boeing Commercial Airplane Company, Seattle, WA, 1981.

[7] Sandberg, G.
A Primer on Relational Database Concepts.
IBM Systems Journal 20(11:23-40, 1981.

[8] Stonebraker, M.
Implementation of Integrity Constraints and Views by Query Modificatioa
In *Proceedings of the 1975 S/GMOD Workshop on Management of Data,* pages 65-78. Association for Computing Machinery, New York, NY, 1975.