CONSTRAINT EQUATIONS: Declarative Expression of Constraints With Automatic Enforcement

Matthew Morgenstern

University of Southern California[1]
Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

Constraint Equations provide a concise declarative language for expressing semantic constraints that require consistency among several relations. Each constraint is independently specified in application based terms and provides a natural extension to the limited semantics captured by typical schemata. Automatic constraint enforcement is accomplished by compilation of the Equations into executable routines, according to the algorithms presented here. A prototype system has shown the viability of this approach. The Equations are more natural and perspicuous than the predicate calculus formulas into which they may be translated. The equivalent of both existential and universal quantifiers are expressible directly in Constraint Equations. Algebraic rules for symbolic manipulation of these Equations allow derivation of new Equations and their logical consequences from existing Equations.

1. Introduction

The integrity and consistency of a database require that a variety of implicit and explicit constraints be maintained among the data. Schema declarations provide a limited set of constraints which supplement the actual data. One may determine from different schemata, for example, the relations which connect attributes or entities, the keys of a relation, or the allowed multiplicity for attributes.

Various data models incorporate certain constraints (eg. existence dependency, uniqueness, etc.) while other constraints are more difficult, or impossible, to represent in the schema. Data models differ with respect to their coverage of these constraints and the defaults they assume. Typically, constraints are embedded in the declaration of the keys of a relation, or in the parent and child segments of a one to many relationship, for example. By making these constraints and dependencies explicit and separable, the perceived differences between these data models may be reduced [Morgenstern81].

Consistency constraints often arise from logical interdependence among several relations, yet such constraints usually go unexpressed due to the absence of a language for representing them in a useful form. While Join Dependencies [Ullman82] could capture a limited class of interrelational constraints, they have been inconvenient for use in applications.

Accurate modelling of an application requires constraints not captured by typical schemata -- such semantic constraints too often are implicit in the protocol of use rather than being explicit. Constraint Equations (CEs) address these issues by providing a declarative representation for inter-relational constraints. CEs easily represent, for example, the application-based constraint that the phone which is to receive messages for a manager is a consequence of who is the assistant for the project run by that manager.

There are several ways in which constraints can arise in databases. They may express application dependent rules for consistency between separately stored data and relationships, thereby helping to define the behavior of the system as the database changes. Rules for derived data also are a form of constraint between the resultant data and the components on which it depends. Whether or not there is a distinction between consistency rules and derived data is largely a matter of which data is stored and one's viewpoint.

Views also involve constraints, since the selection and possible transformation of the underlying data establishes constraints between the base data and the information presented in the view. In the future, "active databases" should provide

[1]This research was supported by the Defense Advanced Research Projects Agency (DARPA) contract MDA-903-81-C-0335. Views and conclusions contained in this paper are those of the author and should not be interpreted as representing the official opinion or policy of DARPA or the U.S. Government.

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984

enhanced user interaction and more responsive interfaces, including capabilities for incremental browsing and for interaction directly with the data presented in views [Morgenstern83].

These capabilities will require that the constraints defining the view be enforceable in both directions wherever possible: in order to support updates to the view directly [Dayal78], and for dynamic maintenance of the view as the underlying data changes and/or as the view definition is modified by the user. Furthermore, transformations from one data model to another can be seen as constraints which define and enforce the mappings between multiple representations [Morgenstern81]. The different types of constraints may warrant different implementations for efficiency, though the underlying concepts are similar.

1.1. EXAMPLE and BACKGROUND

Constraint Equations (CEs) provide a concise declarative language for expressing invariant relationships which must hold among specified data objects and component relationships. This is preferable to writing procedural code to express and enforce the constraints. Furthermore, the declarative Constraint Equations have an executable interpretation, and can be compiled directly into routines for automatic maintenance of the Constraints. This case of automated generation of programs from constraint specifications has been demonstrated in the prototype implementation.

The declarative nature of Constraint Equations together with their executable interpretation have an analogy with algebraic equations. For example, the equation X = Y + Z is a declarative statement of an equivalence between the expressions on either side. If this is to be treated as a constraint which is to be maintained by the system, then there is an executable interpretation which may be thought of as two condition-action rules: (1) if Y and/or Z change, then revise the value of X accordingly, and (2) if X changes, select between the alternatives of disallowing the change, revising Y, or Z, or both.

The following example of a Constraint Equation specifies that the Projects of a Manager are to be the same as the set of Projects which his/her Employees work on.

```
MANAGER.PROJECT == MANAGER.EMPLOYEE.PROJECT
```

Here the dot "." may be thought of as standing in for the relationship between the entities (objects) appearing on either side of it. In general, the dot allows a form of ellipsis in which the attribute or entity name may be omitted.

The CE may be read from left to right as "the Manager's Projects are the same as the Manager's Employee's Projects." Since MANAGER begins the path of associations on both sides of the CE, it serves as the *Anchor* or common binding for both paths. The CE is to hold for each instance of Manager in the database.

Each side of the CE describes a sequence of relations from the Anchor on the left to the *Target* object on the right of the path. There will be a set of one or more Target instances associated with one Anchor instance by these relationships. This CE says that the sets of Projects that arise from both sides must be equal, and that this must be true for each Manager.

Constraint Equations express semantics of an application modularly and in a concise form which is indicative of their meaning. The modularity is two-fold: additional CEs can be added incrementally in any order, and each such Equation is specified non-procedurally with respect to a local context of relevant data objects and relationships. That CEs represent an important class of semantics is evidenced by their contribution in extending KL-ONE, a semantic network used for knowledge representation in artificial intelligence research [Morgenstern84].

One of the earliest approaches for augmenting schema descriptions with additional semantics were the database procedures of the CODASYL network database [Wiederhold77]. These provided a means of executing procedures to derive data or perform other actions. The Query by Example relational system has triggers which can be used to invoke integrity rules [Zloof82]. Other efforts have provided additional primitives in the data model representation [Hammer&McLeod81], utilized semantic nets and/or attached procedures, or a combination of these (see the TAXIS system in [Mylopoulos80]). Recent extensions to the INGRES database system use a QUEL-like syntax for expressing rules [Stonebaker83]. A rule is selected when it syntactically matches a user query, which is then modified by the rule. Also relevant are studies of constraint-based systems, including [Borning79], [Goldstein80], [Sridharan80], [Stefik80], and [Sussman80].

The Constraint Equation facility has been implemented in prototype form utilizing the Information Management system [Balzer83, Morgenstern83] as the initial testbed. Non-trivial hand written code for constraint maintenance has been replaced by routines which were automatically generated from the CEs. In the future, a transportable Constraint Equation package may be provided for use in other selected host environments which support database triggers.

Constraint Equations are defined in sections 2 and 3, together with their translation into predicate calculus. A symbolic algebra for manipulating CEs is presented in section 4, while section 5 describes the update semantics and the algorithms for constraint enforcement. Section 6 shows the further expressiveness of CEs, including existential and universal quantifiers, alternative update semantics, and cardinality based comparators in lieu of set equality in the CE.

2. Constraint Specification and Connection Paths

Each side of a Constraint Equation is a *Path Expression*, which is an abbreviated representation for a sequence of data objects and relationships from the schema for the application. The elided components are determined by comparing the abbreviated path with the database schema.

Here we utilize an Entity based schema for our examples, though the approach also is applicable to other data models. An Entity has a designated type. Its attributes may be single or multi-valued, and may refer to literal values or to other typed entities. For now we assume that attributes are binary relationships, though the methods extend to n-ary relations. Consider the following partial Entity schema, where the indicated attributes of Manager are the only ones directly relating it to an Employee and to a Project. (The -->> symbol denotes a multi-valued attribute.)

```
MANAGER   Entity
          OVERSEES -->> PROJECT
          MANAGES  -->> EMPLOYEE

EMPLOYEE  Entity
          WORKSON -->> PROJECT
```

The translation of the above Constraint Equation from abbreviated Path Expressions into the fully expanded *Connection Paths* is shown here:

```
MANAGER.PROJECT  ==  MANAGER.EMPLOYEE.PROJECT

[ (MANAGER) OVERSEES (PROJECT) ]  ==
[(MANAGER) MANAGES (EMPLOYEE) WORKSON (PROJECT)]
```

Each Path Expression is translated from its abbreviated form by determining whether each of its explicit components is the name of an entity type or the name of a relation, and in the process filling in the possibly omitted entity or relation/attribute name. Elision of a longer sequence of entity and relation names could be allowed when there is no ambiguity. The fully expanded sequence of associations from the Source to the Target is called a *Connection Path*. The leftmost component of the Path Expression must be an entity type, which represents the Source of the path. When the translation is completed it must end on the right with an entity type, which is the Target. The CE is considered ill-formed if there is ambiguity in the translation.

In general, a simple *Connection Path* is a sequence of the form:

$$[\ (E0)\ R1\ (E1)\ R2\ \ldots\ RN\ (EN)\ ],$$

where Ei denotes an entity (object) type, and Ri denotes a (binary) relationship/attribute from E_{i-1} to Ei. (Entities are shown in parentheses when there may be ambiguity between the names of entities and relationships.)

A Connection Path defines a derived relation Rcp in terms of a sequence of Joins over relations Ri. For each pair of relations Ri and R_{i+1} shown above, the natural join is taken with respect to their common domain (Ei). The result is projected onto the domains E0 and En, which are the *Source* and *Target* domains, respectively, of the Connection Path. For a Constraint Equation, in which both sides begin on the left with the same domain E0, we refer to E0 as the *Anchor*, since it anchors the CE with a common binding for both paths.

When a set of instances is provided for domain E0 (or En) of the derived relation Rcp, the Connection Path defines the selection of tuples from Rcp based on these instances, and their projection onto the other domain -- thus providing a mapping from one set of instances to a related set of instances. In particular, for an instance of the Anchor E0, the Connection Path provides a mapping to a set of Target instances, En.

A *composition* of Paths is itself a new Connection Path. Pairwise composition corresponds to the natural join over the Target domain of the left path and the Source domain of the right path. Thus a Connection Path, or composition of subpaths, can be used wherever a relation is used in a Constraint Equation.

3. Formal Interpretation of Constraint Equations

Constraint Equations can be viewed as a compact shorthand for a class of predicate calculus constraints that are useful for database applications.

Consider first the following general Connection Path. Each relation may be viewed as a binary predicate, such as R1(E0, E1). The overall Path represents a derived relation, and is expressed in predicate calculus with set notation following the equality:

$$[\ (E0)\ R1\ (E1)\ R2\ \ldots\ Rn\ (En)\ ]\ =$$

$$\{\ (E0,\ En)\ \mid\ \exists E1,E2,\ldots,E_{n-1}\ [\ R1(E0\ E1) \wedge R2(E1\ E2) \wedge \ldots \wedge Rn(E_{n-1}\ En)\ ]\ \}$$

All entities other than the Source and Target are existentially quantified along the Path. A Constraint Equation consists of two such Paths. Consider the following CE expressed in abbreviated form and then expanded into the full Connection Paths:

$$E0.E1\ ==\ E0.E2.E3$$

$$[\ (E0)\ R1\ (E1)\ ]\ ==\ [\ (E0)\ R2\ (E2)\ R3\ (E3)\ ]$$

Expressing the equality of the two Connection Paths in terms of predicate calculus yields:

$$\{\ (E0\ E1)\ \mid\ R1(E0\ E1)\ \}\ =$$

$$\{\ (E0\ E3)\ \mid\ \exists E2\ (\ R2(E0\ E2) \wedge R3(E2\ E3)\ )\ \}$$

An alternative formulation emphasizes the fact that a CE may be thought of as being implicitly iterated over the instances of the Anchor E0. This viewpoint is valuable for understanding

Constraint Equations, and is utilized later when expressing the Path Quantifiers.

$$\forall E0\ \{\ E1\ |\ R1(E0\ E1)\ \}$$
$$=\ \{\ E3\ |\ \exists E2\ (\ R2(E0\ E2)\ \wedge\ R3(E2\ E3)\ )\ \}$$

Here, each E0 instance serves both sides as a common binding for the Anchor. And each side defines a mapping to a set of Target instances -- the Target sets for the left and right sides being {E1} and {E3}. The CE constrains these two sets to be equal for any such Anchor instance. The equality comparison can be replaced by a superset/subset comparison, as discussed further in section 6.

The equality based CE also may be expressed without set notation as:

$$\forall E0,E1\ [\ R1(E0\ E1)\ \ \langle ==\rangle$$
$$\exists E2\ (\ R2(E0\ E2)\ \wedge\ R3(E2\ E1)\ )\ )$$

4. Symbolic Transformation of Constraint Equations

An algebra for symbolic manipulation of these Constraint Equations enables the derivation of new related Equations from one or more existing Equations. This makes possible symbolic analysis of the consequences of constraints, as well as the derivation of alternative representations. Selected theorems are presented here with summaries of the proofs.

Consider Connection subpaths (or Path subexpressions) P1, P2, P3, and P4 such that P3 may be composed on the right of P1, and P4 may be composed on the left of P1. The first theorem shows that composition preserves the Constraint Equation.

THEOREM #1: **Composition for Constraint Equations**

Given Constraint Equation

$$P1\ ==\ P2$$

Then

$P1\ P3\ ==\ P2\ P3$ composition on the right
and
$P4\ P1\ ==\ P4\ P2$ composition on the left.

Proof: The first derived expression above follows from the fact that each tuple in the derived relation for subpath P1 has a corresponding tuple in P2, and vice versa. Thus the same tuples from P3 will be selected on both sides of the CE based on the join with either P1 or P2. A similar argument confirms the second result.
∎

The substitution of one Constraint Equation into another preserves the set equality of the original CE:

THEOREM #2: **Substitution**

Given Constraint Equations

$$P1\ ==\ P2\ P3\ P4 \quad \text{and}$$
$$P3\ ==\ P5$$

Then

$$P1\ ==\ P2\ P5\ P4$$

Proof: For CE $P3\ ==\ P5$ compose P2 on its left, and compose P4 on the right, yielding $P2\ P3\ P4\ ==\ P2\ P5\ P4$. The theorem follows.
∎

The next two theorems lead to the result for transposing a path component from one side of the CE to the other (Theorem 5).

Path Pj with Source Xi and Target Xj is written $[(Xi)\ Pj\ (Xj)]$. Its *path inverse*, denoted Pj^{-1}, is written $[(Xj)\ Pj^{-1}\ (Xi)]$, which is just the set $\{(Xj\ Xi)|Pj(Xi\ Xj)\}$.

THEOREM #3: **Path Inverse**

For subpaths P1 and P2

$$(P1\ P2)^{-1}\ ==\ P2^{-1}\ P1^{-1}$$

Proof: Given subpaths $[(E0)\ P1\ (E1)]$ and $[(E1)\ P2\ (E2)]$, then $(P1\ P2)^{-1} =$
$\{(E2\ E0)\ |\exists E1\ P1(E0\ E1)\ \wedge\ P2(E1\ E2)\} =$
$\{(E2\ E0)\ |\exists E1\ [P2^{-1}(E2\ E1)\ \wedge\ P1^{-1}(E1\ E0)]\ \}$,
which is just $(P2^{-1}\ P1^{-1})$
∎

The notation, $P1|_{P2}$ expresses the restriction of subpath P1 to those tuples for which the Target values are also Source instances of P2. This serves to exclude dangling tuples of P1 relative to P2 on the right. It is essentially the same as the semijoin [Ullman82] of P1 relative to P2 -- no ambiguity arises here even if both the Source and Target of P1 have the same name, such as for a Manages relation from Employees to Employees. $P1|_{P2}$ may be expressed as the set $\{(X\ Y)\ |\exists Z\ [P1(X\ Y)\wedge P2(Y\ Z)]\}$. Similarly ${}_{P3}|P1$, denotes restricting the tuples of P1 to those for which the Source is also a Target instance of P3 -- ie. excluding dangling tuples of P1 relative to P3 on the left. The following shows the result of composing a subpath with its inverse.

THEOREM #4: **Path Cycle**

Given subpaths P1, P2, and P3, such that
(P1 P2) and (P3 P1) are valid compositions, then

$$P1\ (P2\ P2^{-1}) \supseteq P1|_{P2}$$

$$(P3^{-1}\ P3)\ P1 \supseteq {}_{P3}|P1$$

Proof: The set $\{(E0\ \ E1')|\ \ \exists E1,E2\ [P1(E0\ \ E1)\ \wedge\ P2(E1\ \ E2)\ \wedge\ P2^{-1}(E2\ \ E1')]\}$ represents the left side of the first expression. Choosing E1' as the value for E1, produces the following subset $\{(E0\ \ E1')|\exists E2[P1(E0\ \ E1')\wedge P2(E1'\ \ E2)]\}$, which is the right side of the first assertion.

If $P2^{-1}$ is single valued, then $\supseteq$ will reduce to set equality. Also, if P2 represents a required relationship for Target entities of P1, then the first restriction on P1 is automatically satisfied. The second expression is proved in the same manner, and the $\supseteq$ reduces to set equality if P3 is single valued.
∎

A path component Pi may be transposed from one side of a CE to the other side. This changes the set equality to set cover, and dangling tuples must be excluded from the result.

THEOREM #5: **Transposition**

Given the Constraint Equation

$$P1 == P2\ P3\ P4\ .$$

Then each of the following are true:

$P1\ P4^{-1}$	$\supseteq$	$P2\ P3\ \vert_{P4}$	Transposition on right
$P2^{-1}\ P1$	$\supseteq$	$_{P2}\vert P3\ P4$	Transposition on left
$P1\ P4^{-1}\ P3^{-1}$	$\supseteq$	$P2\ \vert_{(P3\ P4)}$	Multiple on right
$P3^{-1}\ P2^{-1}\ P1$	$\supseteq$	$_{(P2\ P3)}\vert P4$	Multiple on left

Proof: Use the first part of Theorem 4 with P1 there replaced by (P2 P3), and P2 replaced by P4, to get $(P2\ P3)\ P4\ P4^{-1} \supseteq (P2\ P3)\vert_{P4}$. Composing $P4^{-1}$ on the right of the given CE P1 == P2 P3 P4 and comparing these derived CEs proves the first assertion. Note that if $P4^{-1}$ is single-valued, then $\supseteq$ will reduce to set equality.

For the second assertion, compose $P2^{-1}$ on the left of the CE given above. Use the second part of Theorem 4, with P3 there as P2, and P1 there as (P3 P4). The superset will reduce to set equality if P2 is single-valued. The third and fourth parts follow from parts 1 and 2 respectively, by renaming subpaths and applying Theorem 3.
∎

4.1. Example

As a brief example of the use of these algebraic operations, consider again the Constraint Equation where Employees Work on Projects, and Managers Oversee those Projects:

```
MANAGER.PROJECT  ==  MANAGER.EMPLOYEE.PROJECT

[ (MANAGER) OVERSEES (PROJECT) ]  ==
[(MANAGER) MANAGES (EMPLOYEE) WORKSON (PROJECT)]
```

We use the first assertion from Theorem 5 for transposing on the right to obtain:

$$[\ (\text{Manager})\ \text{OVERSEES}\ (\text{Project})\ \text{WORKSON}^{-1}\ (\text{Employee})\ ] \supseteq [\ (\text{Manager})\ \text{MANAGES}\ (\text{Employee})\vert_{\text{WORKSON}}\ ]$$

In this CE the Manager-to-Employee association via Projects yields a superset of the Employees MANAGEd by that Manager -- when restricted to those Employees who directly Workon some Project. (Alternatively stated, for some manager Fred, the set of Employees who work on Projects which Fred oversees is a superset of those Employees Fred Manages directly -- where this set considers only Employees directly working on Projects.)

This transformation of the original CE makes two consequences more apparent. The $\supseteq$ comparator highlights the fact that if several Employees work on a Project, then not all of them need report to the same Manager. If only one Employee works directly on a Project, then $\supseteq$ becomes set equality. And the restriction on Employee highlights the fact that the Manager also may manage other Employees who do not directly work on an existing Project, such as secretaries.

This algebra for Constraint Equations, part of which is presented here, provides a useful means of analyzing the consequences of constraints, reasoning about the application domain, and deriving related Constraint Equations.

5. Update Semantics and Automatic Constraint Enforcement

When changes occur to the database, one or more Constraint Equations may be affected. The constraints are automatically enforced with respect to these changes. In some cases, the constraint may require rejection of the initial database change. Usually, however, the constraint may be satisfied by making consequential changes which depend upon the initial change.

The Constraint Equation specifications are used by the CE Compiler to automatically generate programs which enforce the constraints. The executable interpretation for a CE is reasonably intuitive, and is detailed in the algorithms below. The enforcement routine will make the compensating changes needed to satisfy the constraint(s) following an initial database change. The normal response for enforcing a CE may be modified for special cases by annotating the CE, as discussed later.

The database system implementation provides *triggers* or demons which are activated when changes occur to specified relationships [Goldman82]. The CE Compiler attaches the enforcement routines it generates to database triggers for each of the relation types that are involved in the Constraint Equation. Thus when an insertion, deletion, or update occurs to any instance of these relations, this enforcement routine is automatically invoked to take the appropriate action.

5.1. Changes to an Entity

When an entity (object) instance is created, deleted, or updated, changes occur to relationships which involve that entity. In particular, creation of an entity places it into a system table or relation. If the entity type has required attributes, then these must be defined with the creation of the entity instance. For deletion of an entity, all attributes and relationships involving this entity instance are deleted also. Updating an

entity actually involves updating the attributes/relations of the entity.

Constraint Equations are activated by changes to relationships and attributes. Thus creation, deletion, or updating of an entity would invoke a CE by virtue of changes to attributes and relationships for that entity instance.

5.2. Changes to an Attribute Relationship

A change to a relationship on one side of a CE usually may be compensated for by a change to the other side of the CE, so as to reestablish satisfaction of the constraint. If there is more than one relation on the other side of the CE, then the one to change must be designated to remove ambiguity. Notationally this is indicated by the "!" symbol to the left of or in place of an attribute or relation name (the "!" is used in lieu of the dot "."). The designated relation can be thought of as a *weak bond*, since it is more readily modified in response to an initial change to the other side of the CE.

As an example, consider the constraint that an Employee's Phone's Backup (the extension which takes messages when the phone is busy or does not answer) is the same as the Employee's Project's Secretary's Phone. This may be expressed in a CE as:

```
EMPLOYEE.PHONE ! BACKUP   ==
        EMPLOYEE.PROJECT.SECRETARY.PHONE
```

The designation of weak bond on the left indicates that if any of the associations on the right changes (eg. a Project's Secretary) then the Backup extension for the Employee's Phone is changed. The absence of a weak bond on the right indicates that a change directly to the relations on the left is *not* allowed *if* it would cause a violation of the constraint. For example, the Employee's Phone could be changed to any other Phone having the same Backup without violating the constraint. Alternative update semantics are specifiable by annotations, as discussed below.

The update semantics are reasonably intuitive when relationships are single valued. If an Employee changes to a different Project, and all the remaining relationships (except the changed relation and the weak bond relation) are single valued, then the Secretary's Phone is clearly defined, and the change of Backup extension for the Employee's Phone is simple.

The potentially multi-valued relationship between Employees and Projects can give rise to a set of changes in other cases. If the Secretary's Phone is changed, then the Backup extension must be changed for the Phones of the (potentially) several Employees on the associated Project(s) (ie. on Projects served by that Secretary, and limited to those Phones having the old Backup number). Also, a change affecting one CE can result in a compensating change which activates other CEs -- thus an initial change can create a wave of propagation through several interconnected constraints [Morgenstern83].

As another example, consider the CE presented earlier where a Manager oversees those Projects which are worked on by his/her Employees:

```
MANAGER ! PROJECT == MANAGER ! EMPLOYEE.PROJECT
```

The weak bond on each side indicates that Projects stay with the Employee if there are any other changes. Thus if a Manager adds a Project, then he adds the Employee(s) who already work on that Project (rather than having his existing employees take on that project).

5.3. Algorithms

The rest of this section presents the details of algorithms for automatic enforcement of Constraint Equations of the type considered so far, and may be skipped on a first reading. The CE takes the following form, where Pi are subpaths and Rj are relations:

```
Anchor . P1 . R0 . P2 . L-Target  ==
Anchor . P3 ! Rw . P4 . R-Target
```

Both sides may be expanded into the following Connection Paths. The right side is reexpressed as the composition of three subpaths for convenience.

```
[ (Anchor) P1 (X1) R0 (X2) P2 (L-Target) ]

 == [ (Anchor) P3 (X3) !Rw (X4) P4 (R-Target) ]

 == [ (Anchor) P3 (X3) ] [(X3) !Rw (X4)]
                         [ (X4) P4 (R-Target) ]
```

The relation Rw is designated as the weak bond by the "!" symbol (or by annotations considered in a later section). A compensating change may be made to Rw when an initial change occurs to the relationships on the other side of the CE.

Absence of a weak bond designation would require rejection of all changes to the other side, unless such a change continued to satisfy the invariant. An initial change also may be rejected when other factors prevent reestablishment of the constraint, as indicated in the algorithms (eg. violation of cardinality restrictions).

Consider an initial change (insertion, deletion, or update) to an instance of the R0 relation, for a pair of X1 and X2 object instances. The side containing R0 (shown on left) is treated as independent and the other side as dependent. (If an Anchor or Target instance participates directly in the changed relationship, then subpaths `P1 X1` and/or `X2 P2` are not needed below.)

Four steps are common to insertion, deletion, and update:

(S1) Locate Anchor instances on the independent side: The subpath `[(Anchor) P1 (X1)]` is used from right to left to locate instances of the Anchor associated with the instance of the X1 object.

(S2) Locate Target instances on the independent side: The subpath `[(X2) P2 (L-Target)]` is used to locate instances of the left Target (L-Target) associated with the instance of the X2 object. If either steps S1 or S2 yield a null set, then no further processing is required, as no complete path connecting an Anchor to a Target instance on the independent side was affected by the change.

(S3) Locate Left Bond objects (X3) on the dependent side: For one instance of the Anchor (from step 1) use the left subpath `[(Anchor) P3 (X3)]` to locate the set of associated instances of X3.

(S4) Locate Right Bond objects (X4) on the dependent side: Traverse the right subpath `[(X4) P4 (R-Target)]` from right to left from each instance of the Target to a set of X4, and take the union of these X4 sets.

Algorithm #1: **Insertion**

Insertion of a new R0 relationship between an instance of X1 and an instance of X2:

For each Anchor instance associated with the change (from step S1), there is to be a connection on the *dependent* side to each potentially new Target instance (from step S2). If such a connection does not already exist, find the sets of Left (X3) and Right Bond (X4) objects that could be involved (steps S3 and S4) -- if either set is empty, create new associations according to other specifications (or via user interaction) to make these sets non-empty; or else disallow the original change to R0.

For each pair of X3 and X4 instances, create a new Rw relationship if one does not already exist. A user provided predicate may restrict this cross product of X3 and X4 instances. (The predicate may request more information from the user.) If Rw is required to be single-valued, then there must be not more than one X4 instance related to each X3. For each Anchor, at least one path must connect to each of the Target instances in order for the change to R0 to be accepted (needed to maintain set equality). Repeat the above steps for each Anchor instance.

Algorithm #2: **Deletion**

Deletion of an R0 relationship between an instance of X1 and an instance of X2:

For each Anchor instance associated with the change (from step S1), consider each newly disconnected Target instance (from step S2) having no other connection(s) on the *independent* side, and for each remove the corresponding connections at the Rw link on the other side: Find Left Bond (X3) and Right Bond (X4) objects (steps S3 and S4), and delete Rw relationships which relate instances from these two sets. Naturally, a required relation/attribute should not be designated as the weak bond. Those X3 involved in Rw deletions are utilized in the Update algorithm, below. Repeat for each Anchor instance.

Algorithm #3: **Update**

Replacing an existing R0 relationship between a pair of instances x1 and x2old (for object types X1 and X2), with a new R0 relationship between x1 and x2new:

For each Anchor instance associated with the change (from step S1), use x1 and x2old for Deletion, Algorithm #2. And for the same Anchors use x1 and x2new for Insertion, Algorithm #1 -- except that a non-empty set of X3 objects found during Deletion are used instead of step S3 (so that the update involves the same Left Bond (X3) objects). Repeat this process for each Anchor instance.

6. Enhanced Expressive Power

Here we describe additional expressive power available with Constraint Equations. In particular we show that the set oriented semantics of CEs naturally admit the expression of both the Universal and the Existential Quantifiers.

In addition, the default update semantics may be augmented by annotations indicating that the choice of weak bond should be conditional on which relation changed. We provide comparators between the two Connection Paths of the CE in addition to set equality. And we allow the normal set operators within Connection Paths. The relevant extensions to the constraint enforcement algorithms are not detailed here.

6.1. PATH QUANTIFIERS

Existential Quantifiers are implicit in Constraint Equations, as seen in the previous section on Formal Interpretation. All intermediate entities along the Connection Path (other than the Anchor and Target) have been existentially quantified for the type of CEs shown above.

Existential Quantification corresponds to the natural interpretation of each side of the CE as being the *union* of the Target instances for an Anchor instance. These different Target instances arise from the existence of different sequences (paths) of intermediate relationships connecting them with the Anchor. It is the union of these Target instances for an Anchor that we find with the Connection Paths used so far. This resultant mapping from Anchor instance to Target set is one way of looking at the Connection Path. It may be seen as a derived relation by taking the cross product of the Anchor with its Target set.

The ability to express the *Universal quantifier* can be important. It is needed for constraints such as: the Projects of a Department are those Projects on which *all* the Employees of that Department work. In other words, *the Projects of a Department are those which are common to every Employee of that Department*. This notion of commonness to all sets of instances arising from a (possibly derived) association is represented as a *Path Intersection Quantifier* "∩/ ". This parallels the default interpretation of a Connection Path as being a union of its Target sets, except that here we take the *intersection over* the Target sets. This example may be represented as:

```
DEPARTMENT.PROJECT  ==
            [DEPARTMENT.EMPLOYEE ∩/ PROJECT]
```

The intersection here is over the sets of Projects that are arise from each of the Employees of a Department. Each Employee works on a set of Projects; the intersections of these sets yields just those Projects that everyone works on in that Department. The CE requires that this resulting set of common Projects is to be equal to the set of Projects which the Department directs.

We expand this CE into a full Connection Path using the previous partial entity schema together with the definition of the Department entity:

```
DEPARTMENT entity
           DIRECTS -->> PROJECT
           EMPLOYS -->> EMPLOYEE
```

```
[ (DEPARTMENT) DIRECTS (PROJECT) ]  ==
[ (DEPARTMENT) EMPLOYS (EMPLOYEE)
          ∩/ (EMPLOYEE) WORKSON (PROJECT) ]
```

Expressing this constraint in terms of sets, we have:

```
∀ DEPARTMENT
{ PROJECT | DIRECTS(DEPARTMENT PROJECT) }
=
{ PROJECT |
  ∃ EMPLOYEE ( EMPLOYS(DEPARTMENT EMPLOYEE) ) ∧
  ∀ EMPLOYEE ( EMPLOYS(DEPARTMENT EMPLOYEE) ⇒
            WORKSON(EMPLOYEE PROJECT) ) }
```

In the second set above, we require that at least one Employee is employed by that Department, and that for a Project to be included in the resulting set, every such Employee of the Department Workson that Project. Note that the existence of least one Employee in the Department is required here to ensure that the predicate calculus Universal Quantifier does not become satisfied for each and every Project just because there are no Employees in that Department! Such concerns are taken care of by the semantics of the *Path Intersection* quantifier.

More generally, a Path Intersection expression such as

```
[ E1 . E2 ∩/ E3 . E4 ]
```

expands to a Connection subpath of the form

```
[ (E1) R2 (E2) ∩/ (E2) R3 (E3) R4 (E4) ] .
```

This represents a derived relation between domains E1 and E4. For an E1 instance, this path yields *those E4 instances which are common to every E2* -- ie. an E4 instance is related to an E1 by this path if this E4 is related to every E2 associated with this E1.

We may formally express this derived relation Rcp(E1, E4) by the following set of pairs. The universal quantifier applies to the entity E2 which immediately precedes the Path Intersection symbol (∩/) in the expressions above. The scope of the universal quantifier is the immediately containing bracketed path expression. The other intermediate objects along the path (here E3) are existentially quantified as usual.

```
{ (E1 E4) | ∃ E2 ( R2(E1 E2) ) ∧
   ∀ E2 ( R2(E1 E2) ⇒
         ∃ E3 ( R3(E2 E3) ∧ R4(E3 E4) )) }
```

Since this represents a derived relation Rcp(E1, E4), the above Path Intersection (the expression from E1 to E4) can be used as part of a larger Path. Thus quantified expressions can be nested within each other.

6.2. ANNOTATION OF A CONSTRAINT EQUATION

There are cases when the change to relations on one side of the CE warrants different responses than those presented earlier. The algorithms stated above presume that a change to one side of a CE may be responded to by a change to the designated weak bond relation on the other side. We can extend the range of possible responses by additional annotations associated with the CE.

These annotations take the form of condition-action rules (production rules) which have proven valuable in knowledge-based Expert systems work in the A.I. community [Hayes-Roth83]. Condition-action rules have the advantage of being modular and easy to specify, yet a set of such rules can express complex relationships and actions. For example, a consistency constraint expressed as a condition-action rule would state the change or combination of changes to the database which serve as the condition for activating the rule. And it would indicate the action to be taken -- typically an expression of how to reinstate consistency. Other forms of action might be to disallow the change, provide information to the user, or invoke a more general procedure to execute an arbitrary action. In fact, the Constraint Equation is directly expressible as a set of such condition-action rules -- one for each relation that may change in the Equation.

Here we use such rules to express exceptions to the primary update rules embodied in the algorithms presented above. The condition indicates the relation change which would activate this exception rule, and optionally, the type(s) of change (insertion, deletion, update). The action or response may be of arbitrary complexity, but primarily is intended to indicate a relation of the CE to which the compensating change should be made -- thus allowing the weak bond relation to be conditional on which change occurred. In addition, if a predicate is provided on the action side, it is taken as the filter which limits the creation of new instances for the selected weak bond relation in the Insertion algorithm above.

The following CE is similar to the one presented earlier, except that here the semantics are that a change of Manager for an Employee changes the Projects the Employee works on. The additional rule overrides the base semantics of the weak bond on the left of the CE. The rule below is invoked when the relationship MANAGER.EMPLOYEE on the left is changed, and the response is to treat the relation EMPLOYEE.PROJECT on the right as the weak bond.

```
MANAGER ! PROJECT == MANAGER ! EMPLOYEE.PROJECT
     except
     MANAGER.EMPLOYEE  ⇒  EMPLOYEE.PROJECT
```

Another example is repeated below with a new response. Here a change to a Project's Secretary would cause the compensating change to be made to the Phone of the old and new Secretaries -- in order that the Backup number (and the phone associated with the Project) stays the same:

```
EMPLOYEE.PHONE ! BACKUP  ==
          EMPLOYEE.PROJECT.SECRETARY.PHONE
     except
     PROJECT.SECRETARY  ⇒  SECRETARY.PHONE
```

6.3. COMPARATORS OF THE CONSTRAINT EQUATION

We first extend Connection Paths by including Set Union, Set Intersection, and Set Difference as means of combining pairs of Connection subpaths to produce a new Connection Path. We require compatibility of the types for the Source domains of each component path, and similarly for the Target domains. (In principle this compatibility restriction could be lifted if we consider the union of two different Source types, or Target types, to be a new type.) Such extended Connection Paths also serve to define a derived (binary) relation, just as for simple Connection Paths.

In a Constraint Equation, the Path on each side gives rise to a set of Target instances for an Anchor instance. So far we have required these two sets to be equal. The natural extension is to allow the *subset* comparators $\subseteq$ and $\supseteq$ between the two Target sets.

We also may specify that the Target sets have elements in common, or that the sets are disjoint. These are special cases of the *Intersection Comparator* -- which is denoted `=m:n=` . Here m is the lower bound and n is the upper bound on the number of elements common to the Target sets on both sides of the CE -- where the restriction is to hold for each Anchor instance. The upper bound defaults to the size ($\underline{S}$) of the larger (or equal size) Target set, and may be potentially different for each Anchor instance. The lower bound defaults to the the smaller of the upper bound or $\underline{S}$. This is consistent with the absence of bounds in the `==` symbol for set equality. Also, `=k=` stands for `=k:k=`.

Hence a constraint that the two Connection Paths, CP1 and CP2, have a non-null intersection for each Anchor instance, is written `CP1 =1:= CP2` . The constraint that the Target sets be disjoint is `CP1 =0= CP2` . The requirement that the intersection of the Target sets has either 1 or 2 members is written `CP1 =1:2= CP2`. In summary, the following comparators specify that for every Anchor instance, the two sets of Target instances satisfy the indicated comparison:

`==`	Set equality
$\subseteq$ $\supseteq$	Subset
`=0=`	Disjoint sets
`=1:=`	Intersection is non-null
`=m:=`	Intersection has m or more members
`=k=`	Intersection has exactly k members
`=m:n=`	Intersection has between m and n members

7. Conclusion

Constraint Equations provide a concise declarative representation for modularly expressing a variety of semantic constraints in application based terms. CEs have a more natural and perspicuous structure than the predicate calculus formulas into which they may be translated. Yet both universal and existential quantifiers are expressible conveniently in CEs, as are criteria for disjointness, common elements, subset relationship, and the typical set union, intersection, and difference. Other extensions will include n-ary relations and predicate restrictions on the paths.

Automatic constraint enforcement is provided in the prototype implementation by compilation of a basic CE specification into the equivalent of condition-action rules. The program which is generated will perform the actions needed to reestablish consistency -- this routine is attached to database triggers which will be activated when the specified relation(s) are changed. Since the activation of a Constraint Equation can result in additional database changes, a chain of activations of several Constraint Equations may arise. The set of such activations defines the consequences of the initial change. Strategies for such constraint propagation and for related optimizations are discussed in [Morgenstern83].

An algebra for symbolic manipulation of these Constraint Equations enables the derivation of new Equations from existing ones, and makes possible symbolic analysis of the constraints and their consequences. The use of the CE algebra to derive alternative representations may prove useful in supporting multiple views and data models.

Acknowledgements

I would like to thank Don Cohen, Jack Mostow, and Neil Goldman for their helpful comments and questions. The predicate calculus formulation and proofs have benefited from Don's useful suggestions.

References

[Balzer83] Robert Balzer, David Dyer, Matthew Morgenstern, Robert Neches, *Specification-Based Computing Environments*, Proc. National Conf. on Artificial Intelligence (AAAI-83), Washington, D.C., August 1983, pp.12-16.

[Borning79] Alan Borning, *Thinglab - A Constraint-Oriented Simulation Laboratory*, Stanford Univ. report STAN-CS-79-746, July 1979, Ph.D. thesis.

[Dayal78] U. Dayal & P.A. Bernstein, *On The Updatability Of Relational Views*, Proc. 4th Very Large Data Base Conf. West Berlin, Sept. 1978.

[Hammer&McLeod81] Michael Hammer, & Dennis McLeod, *Database Description with SDM: A Semantic Database Model*, ACM Trans. on Database Syst., v.6, no.3, Sept 1981, pp.351-386.

[Goldman82] Neil M. Goldman, *AP3 Reference Manual*, June 1982, USC Information Sciences Institute, Marina del Rey, CA.

[Goldstein80] I.P. Goldstein & D.G. Bobrow, *Descriptions for a Programming Environment*, Proc. First Annual Conf. Nat'l Assn for A.I. (AAAI-80), Stanford, CA, August 1980.

[Hayes-Roth83] Fredrick Hayes-Roth, Donald Waterman, & Douglas Lenat, eds., *Building Expert Systems*, Addison-Wesley Pubs., 1983.

[Morgenstern81] Matthew Morgenstern, *A Unifying Approach For Conceptual Schema To Support Multiple Data Models*, Second Int'l Conf. on Entity-Relationship Approach, Washington, D.C., October 1981, pp.281-299.

[Morgenstern83] Matthew Morgenstern, *Active Databases As A Paradigm For Enhanced Computing Environments*, Ninth Int'l Conf on Very Large Data Bases, Florence. Italy, Oct 1983, pp.34-42.

[Morgenstern84] Matthew Morgenstern, *Constraint Equations: A Concise Compilable Representation for Quantified Constraints in Semantic Networks*, AAAI-84 National Conference on Artificial Intelligence, Austin, Texas, August 1984.

[Mylopoulos80] John Mylopoulos, Philip .A. Bernstein, & Harry K.T. Wong, *A Language Facility for Designing Database-Intensive Applications*, ACM Trans. on Database Syst., v.5, no.2, June 1980, pp.185-207.

[Sridharan80] N.S. Sridharan, *Representational Facilities of AIMDS: A Sampling*. Dept of Computer Science, Rutgers Univ, New Brunswick, N.J., Report RU-CBM-TM-86, May 1980.

[Stefik80] Mark Stefik, *Planning with Constraints (Molgen: Part 1)*, Artificial Intelligence Journal, vol.16, 1980, pp.111-140."

[Stonebraker83] Michael Stonebraker, et.al., *Implementation of Rules in Relational Data Base Systems*, Univ. of California, Berkeley, CA., Electronics Research Lab, Memo No. UCB/ERL 83/10. June 13, 1983, 10pp.

[Sussman80] Gerald Jay Sussman and Guy Lewis Steele, Jr, *CONSTRAINTS -- A Language for Expressing Almost-Hierarchical Descriptions*, Artificial Intelligence Journal, v.14, 1980, pp.1-39.

[Ullman82] Jeffrey D. Ullman, *Principles Of Data Base Systems*, 2nd ed., Computer Science Press, 1982, 484pp.

[Wiederhold77] Gio Wiederhold, *Database Design*, McGraw Hill, 1977.

[Zloof82] M.M. Zloof, *Office by-Example: A Business Language that Unifies Data and Word Processing and Electronic Mail*, IBM Systems Jour., 21,3, 1982