Constraint Equations: A Concise Compilable Representation for Quantified Constraints in Semantic Networks

Matthew Morgenstern

Information Sciences Institute[1]
University of Southern California
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. The constraints provide a natural addition to semantic networks, as shown by an extension to the KL-ONE/NIKL representation language. The Equations have a more natural and perspicuous structure than the predicate calculus formulas into which they may be translated, and they also have an executable interpretation. Both universal and existential quantifiers are expressible conveniently in Constraint Equations, as are cardinality quantifiers and transitive closure. For a subclass of these constraints, a prototype compiler automatically generates programs which will enforce these constraints and perform the actions needed to reestablish consistency.

1. INTRODUCTION

Constraint Equations (CEs) provide a concise declarative language for expressing a class of invariant constraints which must hold among chains or sequences of relationships. The declarative Constraint Equations have an executable interpretation, and can be compiled directly into routines for automatic maintenance of the Constraints. This is preferable to writing procedural code to express and enforce these constraints. The prototype implementation has demonstrated such automated generation of programs from CE specifications.

The declarative nature of Constraint Equations and 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 which a Manager has a responsibility for 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 objects or entities appearing on either side of it. In general, the dot allows a form of ellipsis in which the relation name or object type may be omitted.

Each side of the CE describes a sequence of relations from the *Anchor* object (here MANAGER) on the left to the *Target* object on the right of the path. There may be a set of one or more Target instances associated with one Anchor instance by these relationships. This CE says that the set of Projects that arise from both sides must be equal, and that this must be true for each Manager. The concept of Path Quantifiers is defined below to provide existential, universal, and cardinality based quantifiers.

A Constraint Equation represents a structured shorthand for a set of condition-action rules -- a fact which is exploited below when extending the range of behaviors describable using these Equations. The operational semantics of these declarative Equations helps address the need for improved facilities to declaratively represent knowledge of the data's semantics. Fikes has noted that "such declarative facilities would reduce the (knowledge representation) designer's dependence on frame actions, and therefore make the resulting implementation more perspicuous and accessible ...." [Fikes81]. Other studies of constraint-based systems include [Borning79], [Goldstein80], and [Sussman80].

The Information Management system [Balzer83] has provided a testbed for prototype implementation of the Constraint Equation facility. Non-trivial hand written code for constraint maintenance has been replaced by routines that were automatically generated from the CEs. Extensions beyond these operational facilities also are presented below.

2. CONSTRAINT SPECIFICATION and CONNECTION PATHS

Each side of a Constraint Equation is a *Path Expression*, which is an abbreviated representation of a sequence of associations from the semantic network model of the application. The nodes of the network are typed and represent objects -- also sometimes referred to as entities or domains. The attributes of the object are treated here as binary relationships to other objects or to literal values. The abbreviated path expression is compared with the semantic network to determine each elided component, which may be either an object or a relation name. The CE is considered ill-formed if there is ambiguity in the translation. The fully expanded sequence of associations is called a *Connection Path*. For example, consider the following entities: (where "-->>" denotes a multi-valued attribute)

[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.

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

EMPLOYEE Entity
         WORKSON -->> PROJECT
```

The abbreviated Path Expressions of a CE are translated into complete *Connection Paths* (second equation) as follows:

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

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

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

```
[ (E0) R1 (E1) R2 ... RN (EN) ] ,
```

where Ei denotes an object (entity) type, and Ri denotes a (binary) relation from E_{i-1} to Ei. (Relations are shown in parentheses when there may be ambiguity between the names of objects and relations.) E0 is the *Source* and En is the *Target* of the Connection Path. In a CE, E0 also is referred to as the *Anchor*, since it anchors the CE with a common binding for both paths. When an instance is provided for domain E0 (or En), the Connection Path defines a mapping from the Source (if E0 was given) to the Target set.

A Connection Path defines a relation Rcp(E0 En) derived from the sequence of component relations Ri by joining them pairwise on their common domains. A *composition* of Connection Paths also is a derived relation, since each subpath can be treated as a relation in the overall path. Thus a Connection Path, or composition of subpaths, can be used wherever a relation appears 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 formulae that are useful for knowledge representation paradigms. Consider the following Constraint Equation and its expansion into Connection Paths:

```
E0.E1  ==  E0:E2.E3

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

Each relation may be viewed as a binary predicate, Ri(Ej, Ek). Since each side of the CE is a derived relation, we obtain the following expression in predicate calculus with set notation:

```
{ (E0 E1) | R1(E0 E1) }  =
{ (E0 E3) | ∃E2 ( R2(E0 E2) ∧ R3(E2 E3) ) }
```

An alternative formulation emphasizes the fact that a Constraint Equation may be thought of as being implicitly iterated over the instances of the Anchor E0. This viewpoint is valuable for understanding CEs, and is utilized later when expressing the Path Quantifiers.[2]

[2] An algebra for symbolic manipulation of CEs is under development -- it is used to analyze the consequences of constraints and to derive new related Equations from existing ones.

```
∀E0  { E1 | R1(E0 E1) }
     = { E3 | ∃E2 ( R2(E0 E2) ∧ R3(E2 E3) ) }
```

Here, each E0 instance serves as a common binding for the Anchor on both sides. Each Connection Path defines a mapping to a set of Target instances -- the Target sets for the left and right sides being {E1} and {E3}. This CE constrains these two Target sets to be equal for any such Anchor instance. (In lieu of equality, there may be a subset or superset comparator, or a common elements (intersection) constraint, denoted =m:n=, requiring that the Target sets have from m to n members in common -- with =0= denoting no common members, ie. disjointness.)

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

```
∀E0,E1 ( R1(E0 E1)  <==>
          ∃E2 ( R2(E0 E2) ∧ R3(E2 E1) ) )
```

4. UPDATE SEMANTICS and AUTOMATIC CONSTRAINT ENFORCEMENT

When changes occur to the data, one or more Constraint Equations may be affected. A compiler-like facility accepts the Constraint Equation specifications and automatically generates maintenance programs which enforce the constraints (currently for existentially quantified constraints). If there is no way of reestablishing the constraint, then the initial change will not be accepted. Usually however, the maintenance routine can execute the consequential changes needed to satisfy the constraint(s).

The system implementation provides triggers or *demons* which are activated when changes occur to specified relationships [Goldman82]. The enforcement routines which are generated by the CE Compiler are attached to the demons for each of the named relations 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.

When an object instance is created, deleted, or updated, changes occur to relationships which involve that object. Deletion of an object causes all its attributes and relationships to be deleted also. Updating an object actually involves updating the relationships of the object. CEs are activated by these changes to relationships.

When a change occurs to a relationship on one side of a CE, a compensating change may be made to a relationship on the other side in order to reestablish satisfaction of the constraint. Since there may be more than one relation on a side, the one to change is indicated by the "!" symbol to the left of or in place of a 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 as a CE:

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

The designation of a 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 as discussed below.

The update semantics are intuitive when relationships are single valued. If an Employee changes to a different Project, and all the 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 a 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 Employee Phones having the old Backup number).

Since the activation of a CE can result in additional changes to relationships, a chain of activations of several CEs may arise. Each such activation serves to propagate the consequences of the initial change [Morgenstern83]. Similar issues regarding constraint propagation arise in truth maintenance systems [Doyle78].

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

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

The weak bond on each side here indicates that Projects stay with the Employee if there are any other changes. Thus if a Manager adds a Project, then he adds those Employee(s) who work on that Project.

4.1. Specialized Update Semantics

The algorithms stated above assume 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. There are cases when a change warrants different responses. We provide this by additional annotations which take the form of condition-action rules or production rules [Hayes-Roth83]

A consistency constraint expressed as a condition-action rule would state the change or combination of changes to the data 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 algorithms. The condition part 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.

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 is invoked when the relationship implied by MANAGER.EMPLOYEE is changed, and the response is to treat the relation EMPLOYEE.PROJECT as the weak bond for this case.

```
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
```

5. ENHANCED EXPRESSIVE POWER

5.1. Path Quantifiers

The set oriented semantics of Constraint Equations can naturally express a spectrum of quantifiers, including existential and universal quantifiers. Existential quantifiers are implicit in CEs, as shown earlier. All intermediate objects along the Connection Path (other than the Anchor and Target) have been existentially quantified for the CEs discussed above. This corresponds to the fact that the path expression on each side of these CEs produces the *union* of the Target instances for an Anchor. The union operation gives rise to the existential quantification over the different sequences (paths) of intermediate objects and relationships connecting the Anchor with the Target.

The ability to express the *Universal quantifier* is needed for a constraint 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* " ∩/ " -- which replaces the implicit union for a path with an explicit *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 associated with the Employees of that Department (since each Employee works on a set of Projects). The CE requires that the 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 object definitions together with the following Department object:

```
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 expression above, a Project is included in the resulting set if *all* Employees of the Department work on 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) ] .
```

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 ($\cap/$) 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.

5.1.1. Spectrum of Quantifiers

The Path Quantifier concept may be extended to provide a spectrum of quantification capabilities ranging from existential to universal quantifiers. In particular, universal quantification required above that E4 be related to *every* E2, whereas existential quantification requires that E4 be related to *at least one* E2 for an Anchor instance.

We define $_{m}\cap/_{n}$ to be a *Limited Path Quantifier*. If it is used in place of the unconditional intersection quantifier $\cap/$ above, it means that an E4 instance is included if it is related (for a given E1) to at least m E2 instances and not more than n E2 instances. We let |E2| denote the size of the set of E2's which are related to the given E1. The upper bound n defaults to this set size |E2|, and may be different for each Anchor instance. The lower bound m defaults to the smaller of the upper bound and |E2| -- so these defaults are consistent with the unsubscripted path intersection symbol $\cap/$.

For example, the constraint that a Department is responsible for helping to direct a Project if at least three employees of that Department are working on the Project, may be written as:

```
DEPARTMENT.PROJECT  ==
            [DEPARTMENT 3∩/ EMPLOYEE.PROJECT]
```

It can be seen that for the previous path from E1, $_{|E2|}\cap/$ is equivalent to the unconditional *Path Intersection* (universal) quantifier $\cap/$, since this explicit lower bound requires that for an E4 to be included in the result, it must be related to *all* E2s of an E1. Furthermore $_{1}\cap/$ is equivalent to the *existential quantifier*, since for an E4 to qualify, it must be related to just *one* or more E2s. Thus we have a spectrum of quantifiers.

5.2. Path Operators and Transitive Closure

The Connection Path on either side of the Constraint Equation may be extended to include Set Union, Set Intersection, and/or Set Difference of a pair of Connection subpaths. These set operators are subject to the restriction that the Source object type for each subpath is the same, and the Target object type for each subpath is the same. When there is a type hierarchy for objects, this restriction is loosened to require just compatibility of object types.

We may view the set operator in either of two ways: as the union (or other set operator) of the Target sets arising from each subpath for a given Anchor instance, or as the union (or set operator) of the relation tuples from each subpath. Since these are equivalent, the compound Connection Paths also define a derived (binary) relation, just as for simple Connection Paths.

Constraint Equations now can represent the *transitive closure*. For example, consider a programming environment where the system keeps track of potential calling relationships between programs (as provided by the Masterscope package of Interlisp [Teitelman]). The CALLS relationship exists between function F1 and those functions Fj for which a calling form appears in the body of F1. Then the relation REACHABLE for function F1 is the transitive closure of the CALLS relation -- ie. all functions called directly by F1 or indirectly Reachable from such called functions. The Constraint Equation representation is:

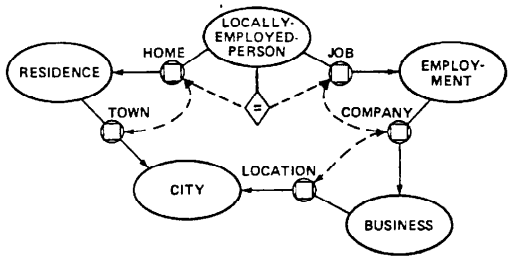
```
FUNCTION ! REACHABLE  ==
FUNCTION.CALLS ∪ FUNCTION.CALLS.REACHABLE
```

This recursive definition takes on the expected meaning due to the executable interpretation of Constraint Equations. In particular, when a new [F1 CALLS F2] relationship is entered, the following responses occur: the right side of the CE causes F2 and all functions Reachable from F2 to be included as Reachable from F1. The resulting change to the REACHABLE relation for F1 causes other activations of this CE for those functions which call F1. In turn this may modify REACHABLE again. The cycle terminates since the union is over a finite number of elements.

5.3. Application to the KL-ONE/NIKL Semantic Network

NIKL is a recently developed knowledge representation system [Bobrow83, Moser83], which is a successor to KL-ONE [Schmolze83], and incorporates ideas from the KRYPTON system [Brachman83]. NIKL also has similarities to the KRL representation language, except that KRL also provides for operational semantics which are specified by collections of attached procedures [Fikes82].

The NIKL semantic network is a taxonomy of *concepts* (intentional objects) which are related by specialization -- indicated by a superconcept (is-a) link. The attributes of a concept are referred to as roles, and may include restrictions such as the number and type of values that may fill the role. Role Constraints (role value maps) are intended as a way of mutually restricting the values that may fill two or more roles. As an example, a Role Constraint for a locally employed person (LE-PERSON) is that his/her home is in the same city as the company which employs the person. The following NIKL diagram from [Moser83] shows this requirement.



A Constraint Equation which represents this constraint is shown in both its abbreviated and complete path forms:

```
LE-PERSON.HOME.TOWN  ==
LE-PERSON.JOB.COMPANY.LOCATION

[ (LE-PERSON) HOME (RESIDENCE) TOWN (CITY) ] ==
 [(LE-PERSON) JOB (EMPLOYMENT) COMPANY
                  (BUSINESS) LOCATION (CITY)]
```

Thus far, universal quantifiers have not been expressible in NIKL. Some consideration had been given to the use of a separate predicate to filter the cross product of values from the several roles, and thereby select those combinations which mutually satisfy the Role Constraint [Bobrow83].

The universal quantifier is captured by Path Intersection in a Constraint Equation. So for example, to express the fact that a person's friends are those people who are friends of *all* his brothers, we write the CE:

```
PERSON.FRIEND == [PERSON.BROTHER ∩/ FRIEND]
```

If the CE did not include the Path Intersection (∩/), then any friend of any brother would be one of the person's friends, rather than requiring friendship with all the brothers in order to qualify.

Thus Constraint Equations overlap with other knowledge representation schemes, and they provide a natural extension to the already rich KL-ONE semantic network.

6. CONCLUSION

Constraint Equations (CEs) provide a concise declarative representation for a commonly occurring class of constraints in which two differently derived sets of instances, and two different chains of relationships, are to be consistent. 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 cardinality quantifiers, transitive closure, and disjointness. Automatic constraint enforcement is provided in the prototype implementation by compilation of a basic CE specification into a program which will perform the actions needed to reestablish consistency.

ACKNOWLEDGEMENTS

I would like to thank Don Cohen, Neil Goldman, Tom Lipkis, and Jack Mostow for their useful comments: the predicate calculus formulation benefited from discussions with Don, Jack suggested the transitive closure example, and Tom offered insight into the current NIKL/KL-ONE system.

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.

[Bobrow83] Rusty Bobrow, *NIKL - A New Implementation of KL-ONE*, Bolt Beranek and Newman, Cambridge, Mass., January 1983, draft.

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

[Brachman83] R.J. Brachman, R.E. Fikes, and H.J. Levesque, *KRYPTON: A Functional Approach to Knowledge Representation*, IEEE Computer, Oct. 1983, pp.67-73.

[Doyle78] Jon Doyle, *Truth Maintenance Systems for Problem Solving*, Masters Thesis, M.I.T., January 1978, A.I. TR-419, 97pp.

[Fikes81] Richard E. Fikes, *Odyssey: A Knowledge-Based Assistant*, Artificial Intelligence Jour., v.16, 1981, pp.331-361.

[Fikes82] Richard E. Fikes, *Highlights from Klone-Talk*, Proc. of the 1981 KL-ONE Workshop, Fairchild Camera Technical Report No.618, May 1982, pp.88-103.

[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.

[Morgenstern83] Matthew Morgenstern, *Active Databases As A Paradigm For Enhanced Computing Environments*, Ninth Int'l Conf. on Very Large Data Bases (VLDB-83), Florence, Italy, Oct 1983, pp.34-42.

[Moser83] M.G. Moser, *An Overview of NIKL, the New Implementation of KL-ONE*, pp.7-26, in Research in Knowledge Representation for Natural Language Representation, October 1983, Bolt Beranek & Newman, Report No.5421.

[Schmolze83] James G. Schmolze and Thomas A. Lipkis, *Classification in the KL-ONE Knowledge Representation System*, Proc 8th Int'l Joint Conf. on A.I., August 1983, Germany, pp.330-2.

[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.

[Teitelman] Warren Teitelman, Interlisp Reference Manual, Xerox Palo Alto Research Center, 1978.