Knowledge-based Integrity Constraint Validation

Xiaolei Qian and Gio Wiederhold
Department of Computer Science, Stanford University

Abstract

One of the important means of specifying the semantics about data is via integrity constraints. Experience has shown that the conventional database approach to integrity constraint enforcement is not successful. In this paper, we demonstrate the feasibility and power of a knowledge-based approach to the efficiency problem of constraint validation. We propose a transformational mechanism which exploits knowledge about the application domain and database organization to reformulate integrity constraints into semantically equivalent ones from which efficient code can be generated.

1 Introduction

The need for integration of knowledge and data into knowledge base systems is widely recognized[12, 15, 19, 20]. Knowledge base systems should provide uniform management of both data and knowledge. To achieve integration we need to generalize, on one hand, the database techniques to cope with the irregularity of knowledge and, on the other hand, to expand the knowledge processing techniques to deal with large amount of data. One of the important means of specifying the semantics of data in a database is via integrity constraints. The existence of a constraint management component in an integrated knowledge base system is essential for the consistency and integrity of data and the validity of knowledge base system.

There are several functions in such a constraint manager. First, it must provide certain means for the user to specify the integrity constraints in the database. The specification capability is usually supplied with some kind of declarative language. Second, the set of constraints specified by the user must be verified to be consistent or satisfiable. The verification of the mutual consistency of constraints is applied once for all possible extensions of the database because it only concerns the static, content-independent properties of the database. This verification process can be performed using theorem proving techniques. However, integrity constraints are intrinsically state-dependent and have to be validated against the database extension whenever state transition happens. This leads to a third validation function in which the key challenge is the efficiency of validation. Finally, the constraint manager has to make decision on what to do when an invalid request for changing database state is encountered. This may include either rejecting the request or making some further state changes to get another valid state.

Research on constraint management in database or knowledge base systems has mainly concentrated on the first two functions[3, 8, 9]. Meanwhile, the efficient validation of constraints against specific database states has been proved to be extremely difficult[1, 5]. Most database researchers have explored the validation problem within the context of relational data model or its extensions[2, 8, 9, 10, 14, 16]. These approaches try to derive efficient constraint validation algorithms from the syntactic structure of the constraint specification. Because of the state-dependent nature of integrity constraints, they can only obtain sub-optimal checking code. Most AI researchers have simply ignored the efficiency issue because they usually deal with a small amount of facts and thus efficiency is of secondary importance.

Stonebraker [16] proposed a validation mechanism in which a request for changing database state is modified at the query language level to contain all the relevant integrity constraints. The query processor will take the responsibility of validation during query processing. This method has been proved to be inefficient because (1) constraints have to be validated as stated by the user, (2) constraints have to be validated at the query processing stage, and (3) constraint validation algorithms do not make use of the knowledge about the specific application domain. The constraint manager has no means to reformulate the constraint checking part of the query to reduce the cost of constraint enforcement. Also the validation effort may have to be duplicated since the constraint manager has to do validation on a query-by-query basis and no knowledge about previous validation activity is maintained.

Bernstein, Blaustein, and Clarke gave an improved scheme in [2] in which some very limited primitive information (in forms of auxiliary aggregate data) is maintained in order to improve the performance of constraint enforcement. However, they only considered a small class of database integrity constraints involving arithmetic comparison operators. Paige applied finite differencing technique to both query optimization and constraint validation in [13]. Although the finite differencing technique is very effective in reducing expensive recomputations to incremental updates, blindly applying it without considering the usage of database sometimes leads to more costly operations.

In this paper, we try to investigate a different approach to integrity constraint validation, one that seeks to exploit knowledge about the application domain to transform user-specified integrity constraints into ones which are syntactically different but semantically equivalent in the sense that they enforce the same condition, and which are cheaper to enforce given the existing database configuration. Rather than seeking the most efficient way of evaluating a constraint as stated, we are looking for methods to reformulate a more effective one. Such reformulation is not simply a syntactic alternative to the original, but instead is a different statement that is only equivalent to the original one under the semantics of application. Hence the reformulation process cannot be made on a purely syntactic basis, it depends on the exploitation of a "knowledge base" about the world of which the database is a model, about the internal organization and access methods, and about the changing state of the database. Conventionally, the implementation of such transformations is beyond the scope of DBMS functions and would have to be encoded by the database designer into triggering procedures. Because of the difficulty of maintaining procedural semantics and the desire for declarative specification of integrity constraints, such a transformation together with the evolution of the knowledge base must be automated.

The paper is organized as follows. In Section 2, an overview is given about the knowledge-based approach we adopt here to the integrity constraint validation problem. In Section 3, we introduce the concept of integrity constraints and briefly classify commonly recognized types of constraints in the context of a relational database. in Section 4, various techniques are discussed to validate constraints more efficiently. An overall architecture of our knowledge base and constraint manager is described in Section 5. Section 6 consists of some example transformation rules used to reformulate constraints. We assume, in the rest of the paper, that the underlying database system is based on relational model and the integrity constraints are expressed in relational calculus, augmented with arithmetic, aggregate, set, and mapping operations. Throughout the discussion, we will take example constraints from the following database scheme about a company database:

EMP(name, birth-date, salary, edept, manager, job)
DEPT(code, name, location, manager, budget, sales)

2 A Knowledge-based Approach to Constraint Validation

Our solution to the efficiency problem mentioned in the previous section is to construct a knowledge base of transformation rules which can be combined to synthesize efficient code for constraint enforcement[6, 7, 11]. Such a knowledge base will consist of a collection of declaratively expressed meta-facts about the application domain and the database implementation, together with a set of rules which transform expensive constraints into cheaper ones. Many types of knowledge can be explored by the constraint manager in order to improve the efficiency, e.g., information about inter-relationships between entities in the application, the database structure and binding, the available access methods and internal file organization, the statistics of database utilization, and the applicability of specific transformation techniques. In this section, we illustrate our knowledge-based approach by examples which explore these kinds of knowledge.

Consider the following example constraint which specifies that none of the employees earn more than their manager:

$$(\forall e \in EMP)(\forall m \in EMP) \\ (e.MANAGER = m.NAME \rightarrow \\ m.SALARY \geq e.SALARY)$$

A naive implementation of enforcement of this constraint specification yields two nested enumerations over the relation *EMP*. The time complexity is thus $O(n^2)$ in the size of *EMP* relation n. With the knowledge that each employee can have exactly one manager and the manager of an employee is represented as a many-to-one database connection *MGR: EMP* $\mapsto$ *EMP* such that:

$$(\forall e \in EMP)(MGR(e) = e' \wedge e' \in EMP \wedge \\ e'.NAME = e.MANAGER),$$

the above constraint can be synthesized into:

$$(\forall e \in EMP)(MGR(e).SALARY \geq e.SALARY)$$

which leads to an implementation with time complexity $O(n)$. With the knowledge about the equivalence between the above assertion and the statement:

$$(\forall m \in EMP)(m.SALARY \geq \\ max\{e.SALARY | e \in EMP \wedge MGR(e) = m\})$$

we may choose to materialize an auxiliary partial mapping: $M : EMP \mapsto REAL$ where

$$(\forall m \in EMP)(M(m) =$$

$$max\{e.SALARY | e \in EMP \wedge MGR(e) = m\}).$$

The availability of such kind of auxiliary information about current database state reduces the cost of constraint validation under insertion to constant $O(1)$. The decision of whether to maintain such information or not can be made by comparing the time saved by not recomputing the maximum employee salary for each manager at insertion time against the time overhead of maintaining such redundant data when a deletion or an update to the employee relation occur. The option of doing so can be expressed in another single transformation rule which takes into account the various update operation frequencies.

Now consider another example, which states essentially the referential integrity between employees and their department, namely that each employee must belong to exactly one department. This constraint is decomposed into two sub-constraints: each employee belongs to at least one department and each employee belongs to at most one department (functional dependency of *EDEPT* on *NAME*). The constraint is expressed explicitly as:

$$(\forall e \in EMP)((\exists d \in DEPT)\ (e.EDEPT{=}d.NAME) \wedge \\ (\forall e' \in EMP)(e'.NAME = e.NAME \rightarrow \\ e'.EDEPT = e.EDEPT))$$

which takes time $O(n \times (n + m))$ where n is the size of EMP relation and m is the size of $DEPT$ relation. Suppose user supplies another constraint saying that the *NAME* attribute of *EMP* relation is the key:

$$(\forall e, e' \in EMP)(e'.NAME = e.NAME \rightarrow e' = e).$$

This key constraint subsumes the second part of the previous constraint. The first part of the previous constraint now becomes:

$$(\forall e \in EMP)((\exists d \in DEPT)(e.EDEPT = d.NAME))$$

and takes time $O(n \times m)$. Furthermore, if there is an inverse mapping or index *M: EMP* $\rightarrow$ *DEPT* where $(\forall e \in EMP)(M(e).NAME = e.EDEPT)$, our constraint can be reformulated into:

$$(\forall e \in EMP)(M(e) \neq NULL)$$

which, with the assumption that index access takes time $O(\log m)$, takes $O(n \times \log m)$. In fact, we can do even better. If EMP relation is clustered with $DEPT$ relation in one file, the constraint is automatically enforced by file organization. The cost of validation is 0!

3 Integrity Constraints

A database is a collection of objects or tuples. Objects are classified into different types. Objects of the same type draw their values from the same domain. An integrity constraint is an abstraction of a logical restriction that objects in the database must obey. The set of integrity constraints specifies the semantics of data in the database and represents knowledge about data.

However, not all constraints in a database represent application semantics. An application has a set of integrity constraints C_a which specify application-dependent semantics. In order to model the application in a database it must be mapped into a fixed set of database structures called a data model. During this mapping process, an additional set of constraints C_s is introduced which are due to the limitations of the types of structures available in a specific data model. Finally, the set of structures has to be implemented on a specific computer with specific implementation techniques. This may introduce further constraints C_i. Thus the set of constraints C in a database is a combination of all the 3 types of constraints: $C = C_a \cup C_s \cup C_i$.

Since some application-dependent constraints can be mapped directly to certain database structure or implementation constraints, $C_a \cap (C_s \cup C_i) \neq \emptyset$. On the other hand, it is usually the case that $C_a \not\subseteq (C_s \cup C_i)$. The set $C_s \cup C_i$ is called inherent constraints in [3, 17] while the rest of C_a: $C_a - (C_s \cup C_i)$ is called explicit constraints. Because the structures in a data model are built into the database system, it is obvious that the structure-dependent and implementation-dependent integrity constraints in $C_a \cap (C_s \cup C_i)$ are efficiently enforced. On the other hand, explicit constraints have to be specified explicitly by user, either as logical assertions or in a particular constraint language, and validated by constraint manager. The more powerful a data model is, the larger the set $C_a \cap (C_s \cup C_i)$ is.

In database systems that are based on relational model, we can think of three types of constraints: (1) domain constraints specify conditions on the values of attributes; (2) connection constraints specify conditions on inter-relation links; and (3) value constraints represent restrictions on the combination of attribute values of a relation, of which dependency constraints are special cases. Only a subset of these kinds of constraints are in $C_s \cup C_i$. For example, constraints about the single-valueness of attributes are enforced by first-normal form relations; functional dependencies are enforced by the uniqueness of keys condition in Boyce-Codd normal form relations; and certain many-to-one connections can be mapped to clustered file implementations. There are still large amount of integrity constraints in C_a that are not enforced by the model.

In the rest of the paper, we'll concentrate on the validation problem of those application-dependent constraints which have to be validated explicitly by the constraint manager, i.e., the set $C_a - (C_s \cup C_i)$.

3.1 Domain Constraints

Domains are sets of values that attributes can take in forming tuples. Domain constraints apply to values of a single type, i.e., values in a single domain. Domain constraints are often used in the definition of a domain to represent the membership condition of the domain. It is actually the means to specify "instance-of" abstractions or classifications. The general form of domain constraints are $(\forall r \in R)(r.A \in D \wedge P(r.A))$, where P is a predicate stating the qualification of the values in domain D of attribute A of relation R. It is equivalent to say that the domain D is defined as $D = \{x|P(x)\}$. The example below shows a domain definition which contains domain constraints.

[Example 1] The domain "DATE" defines a set of structured values each of which consists of three subfields: year, month, and day. There are constraints on both the ranges of values of subfields and the possible combinations of these values.

```
DOMAIN Date
  DESCRIPTION
    Day : integer
    Month : integer
    Year : integer
  CONSTRAINT
    $(Day \geq 1) \wedge (Day \leq 31) \wedge (Month \geq 1) \wedge$
    $(Month \leq 12) \wedge (Year \geq 0) \wedge (Year \leq 9999) \wedge$
    $(Month \in \{4,6,9,11\} \rightarrow Day \leq 30) \wedge$
    $(Month = 2 \rightarrow Day \leq 29)$
END Date
```

3.2 Connection Constraints

Connection constraints denote relationships among objects of one or more compound domains (relations). These constraints restrict the way that objects in these domains can be connected together. A connection constraint $P(D_1, D_2, ..., D_n)$ which relates relations or composite domains $D_i, i = 1, ..., n$ specifies that only a subset of the cartesian product $D_1 \times D_2 \times ... \times D_n$:

$$\{d_1...d_n |\ d_1 \in D_1 \wedge ... \wedge d_n \in D_n \wedge P(d_1, ..., d_n)\}$$

can be part of a valid database extension. In other words, the following statement is always true:

$$(\forall d_1 \in D_1)...(\forall d_n \in D_n)(P(d_1, ..., d_n)).$$

Connection constraints can also be further subdivided into cardinality constraints[4, 18] and value constraints. A cardinality constraint constrains the number of tuples in an inter-relation connection. The constraint specifies a range of allowable numbers for each type of tuples in the connection. In relational model, connections are represented by matching domains of relations. We can characterize a connection among attributes $X_i, i = 1, ..., n$ of relations $R_i, i = 1, ..., n$ by $Conn(R_1 : X_1, ..., R_n : X_n)$. A cardinality constraint on this connection can be expressed as:

Let $VAL = \bigcup_{i=1}^{n} \{v|(\exists r_i \in R_i) \wedge r_i[X_i] = v\}$, then
$(\forall i \in \{1, ..., n\})(\forall v \in VAL)$
$(size\{r|r \in R_i \wedge r[X_i] = v\} \in \{l_i, ..., m_i\})$

where $l_i \leq m_i \in \aleph, (i = 1, 2, ..., n)$ and $\aleph$ is the set of non-negative integers.

A cardinality constraint is called total with respect to a domain D if every object in the domain must be associated with at least one connection. This means that the cardinality of domain D in the constraint specification must be greater than 0. Otherwise the constraint is called partial with respect to D, in the sense that an object may have no associations with others.

[Example 2] A binary cardinality constraint is a special case and has been discussed in great detail in literature[4]. Followed is a table of commonly recognized types of cardinality constraints on binary connections $Conn(R_1 : X_1, R_2 : X_2)$:

	Total (wrt R_1)	Partial (wrt R_1)
One-to-One:	$l_1 = m_1 = 1$	$l_1 = 0, m_1 = 1$
Many-to-One:	$l_1 = 1, m_1 = \infty$	$l_1 = 0, m_1 = \infty$

3.3 Value Constraints

Inter-domain value constraints specify value dependencies among objects in several domains. The value dependency is usually expressed as logical relationships among functions of object values. A value constraint on objects of domains $D_1, D_2, ..., D_n$ has the general form:

$$P(f_1(D_1, ..., D_n), f_2(D_1, ..., D_n), ..., f_m(D_1, ..., D_n))$$

where P is a predicate and $f_i, i = 1, ..., m$ are functions. The functions can be either system-defined or user-defined, numerical- or non-numerical-valued, and intra- or inter-domain functions.

An important special case of the inter-domain value constraints is functional constraints which completely determine the value of the object being constrained. In a functional constraint, P is the identity predicate '=' and has the general form: $D = f(D_1, D_2, ..., D_n)$. Various functional constraints can be established among domains. The functions used to represent the functional constraints are actually mappings among domains. New concepts (domains) may be defined which are arbitrary functions of existing concepts (domains). A lot of properties of these mappings can be examined to obtain knowledge. Distinctions can be made between total versus partial mappings, onto versus into mappings, or one-to-one versus many-to-one mappings, etc. The knowledge about these properties can be used in domain constraint propagation as well as inter-domain constraint enforcement.

Those non-functional constraints are called relational constraints. Relational constraints describe value relationships among object values. However, these object values cannot be determined completely from the constraints.

[Example 3] Derived Attributes. We may want to have an attribute for each department to represent the total number of employees in that department:

$$(\forall d \in DEPT)(d.\#EMP = size\{e | e \in EMP \wedge e.EDEPT = d.NAME\})$$

$\#EMP$ is a derived attribute. Whether it is materialized in the database or not, the constraint manager has to keep consistency between it and the EMP relation.

[Example 4] Virtual Attributes. Suppose the company has several departments in Europe and the staff in those departments would prefer using pound instead of american dollar as the unit of departmental sales attribute. Now we can define a view *V-DEPT* with a virtual attribute *SALES* on top of the real relation *DEPT* such that:

$$(\forall d' \in V\text{-}DEPT)(\exists d \in DEPT)(d'.SALES = d.SALES \times rate \wedge d'[U - \{SALES\}] = d[U - \{SALES\}])$$

where U is the set of attributes of *DEPT* and *rate* is the exchange rate between pound and dollar. Only one relation is materialized and the other one must always be computed according to the above constraint.

4 Validation Techniques

Various techniques can be explored to reduce the cost of constraint validation. In this section, we illustrate several common techniques used by our constraint manager to reformulate constraint.

4.1 Finite Differencing

As we pointed out earlier in the paper, no improvement on validation efficiency can be achieved for integrity constraints which relate single objects together. It is those constraints relating groups of objects in their entirety that give the potential of improvement. Generally the validation of such type of constraints is costly, i.e., at least $O(n)$ where n is the size of the group. Finite differencing is a technique in which costly recomputations are replaced by incremental updates, which can often be done in constant time[13].

Suppose that there is a functional constraint $x = f(y_1, ..., y_n)$. One method of maintaining this constraint is to recompute x whenever any of the parameters $y_i, i = 1, ..., n$ changes. The change to any of $y_i, i = 1, ..., n$ is actually due to object-wise operations such as insertions or deletions of single objects into relations or redefinitions of connections for single objects. Such a "small" change to a parameter of f often results in a corresponding "small" change to the value of f. Finite differencing refers to the detection and exploitation of this situation.

[Example 5] Suppose that x is equal to the set of employees in the department whose manager is *Smith*:

$$x = f(EMP, MANAGER, Smith) = \{e \in EMP | e.MANAGER = \text{"Smith"}\}.$$

Here *EMP*, *MANAGER*, and *Smith* are the parameters of the constraint; *MANAGER* is a mapping whose domain is a subset of *EMP*. *Smith* is an element of the range of *MANAGER*. Suppose *EMP* is updated by the insertion of a tuple for *Brown*. The naive validation of x would recompute the set former. However the value of x can be updated simply by inserting the tuple for *Brown* into x if and only if *Brown*'s manager is *Smith*. In such a situation we say that f is continuous with respect to the operation of tuple insertion to *EMP*. It is easy to see that f is continuous with respect to tuple deletion to *EMP*, or a redefinition of *MANAGER* on a single tuple in *EMP*. However f is not continuous with respect to any changes to *Smith* or to *MANAGER* as a whole.

4.2 Store versus Compute

Even for relational constraints where the value of f is not explicitly required in the database, naive implementations of inter-domain constraint validation require accessing groups of tuples to evaluate the constraints at each time the tuples in a connection are changed. By keeping some relevant information (e.g., the value of f) in the database, it is possible to transform the access operation of groups of tuples to an access operation to single piece of information. In order for this transformation to be cost-effective during the constraint validation process, the maintenance cost of the extra information must be lower than the cost of actually evaluating the constraint for each update.

Such "store versus compute" decision is closely related to the access pattern of the database as well as the cost of recomputation. For expensive computation such as reduction operations of groups of tuples, good candidates for the "store" strategy are those which are continuous with respect to tuple-wise changes in parameters because small changes in parameters are reflected to incremental changes to the stored values.

[Example 6] Suppose that a relational constraint restricts the sum of employee salaries in any department to be less than the department budget:

$$(\forall d \in DEPT)(d.BUDGET > sum\{e.SALARY | e \in EMP \wedge e.EDEPT = d.NAME\}).$$

A direct validation of it would be to recompute

$$f(d) = sum\{e.SALARY| \\ e \in EMP \wedge e.EDEPT = d.NAME\}$$

each time a change is made to EMP. Since $f(d)$ is continuous with respect to changes in EMP, storing the value of f as a mapping $f : DEPT \mapsto REAL$ and incrementally updating it by finite differencing will often results in an asymptotic improvement in performance.

When x is not scalar-valued, the cost of maintaining $x = f(y_1, ..., y_n)$ increases dramatically such that it may outweigh the cost of evaluating f. In such cases, storing x is beneficial only when $y_i, i = 1, ..., n$ are rarely changed and x is accessed frequently. Generally, the "store versus compute" decision depends on the cost of both constraint validation and query processing.

4.3 Discontinuity Removal

Now suppose that finite differencing is not possible since discontinuous changes to a parameter occur. Another alternative would be to remove discontinuity, in which, by the introduction of auxiliary knowledge, changes to a parameter of a function formerly discontinuous becomes continuous. Such auxiliary knowledge could be some internal parameters, functions, or data structures.

[Example 7] Consider the integrity constraint that the number of employees in administrative departments are restricted to be no more than 200:

$$ADM = \{PERSONNEL, PAYROLL\} \\ size\{e \in EMP|e.EDEPT \in ADM\} \leq 200.$$

The constraint is continuous with respect to object-wise changes in *EMP* and redefinition of *EDEPT* for single element in *EMP* but is not continuous with respect to changes in ADM, e.g., adding a new department to it. This can be remedied by introducing as auxiliary information a mapping $f : DEPT \mapsto INTEGER$ where

$$(\forall d \in DEPT)(f(d) = \\ size\{e|e \in EMP \wedge e.EDEPT = d.NAME\}).$$

If y is inserted or deleted from ADM, $f(y)$ is added or subtracted accordingly from the stored size value. Of course this is correct only when an employee can work in at most one department. Otherwise an auxiliary mapping *M: DEPT* $\mapsto$ *set of EMP* would have to be introduced.

4.4 Algebraic Properties

Lets reconsider the constraint $x = f(y_1, ..., y_n)$. The function f may be a composition of several operators. The properties of these operators can help improve the performance of the constraint validation. The constraint manager can apply the knowledge about these properties, such as distributivity, commutativity, and associativity to reformulate f such that it is more efficiently evaluated.

[Example 8] Suppose the company entity has a derived attribute $\sharp E$ which is the total number of employees in the company. The *EMP* domain has been specialized into several sub-domains: *ENGINEER*, *MANAGER*, and *SALESMAN*. The constraint is that

$$\sharp E = sum(size(ENGINEER), size(MANAGER), \\ size(SALESMAN))$$

Applying the techniques discussed before, we would have three pieces of information for the current sizes of the three subsets. Realize that the above assertion is equivalent to:

$$\sharp E = size(ENGINEER \cup MANAGER \cup SALESMAN)$$

we could maintain only one piece of data for the current size of *EMP* and synthesize the tuple-wise operations to all the subsets in such a way that the value of that piece of data is consistently updated.

We have discussed how various techniques can be applied to save constraint validation cost. For complex constraints, application of multiple transformation rules may be necessary before the constraints are actually synthesized into efficient code. A transformation may not be beneficial by itself but can introduce opportunity of optimization.

5 General Model of Constraint Validation

The problem of integrity constraint validation can be stated as follows. Let C be the set of integrity constraints, E be the set of all possible database extensions, $E' \subseteq E$ be the set of all valid database extensions such that the constraints in C are satisfied, O be the set of database operations, $O' \subseteq O$ be the set of database operations that change the database state, and a special operation $o_I \in O$ be the identity operation which, when applied to a database, does not result in any state transition. Given C, a particular database extension $e \in E'$, and an operation $o \in O'$, find out a sequence of database operations in O which, when applied to $e \in E'$ together with o, leads to another valid database state $e' \in E'$. To express it more formally, let $o(e, d)$ denote the resulting database extension by applying operation o to object d in database extension e and $o_2(o_1(e, d_1), d_2)$ denote the composition of operation o_1 followed by o_2 applied to extension e, the task of the integrity constraint manager is to prove or disprove the statement

$$(\exists o_1, ..., o_n \in O)(e \in E' \wedge o \in O' \wedge \\ o_n((...o_1(o(e, d), d_1), ...), d_n) \in E')$$

and if the statement is true, the constraint manager is also responsible for actually finding the sequence of operations. In this paper, our set of operations O' will only include insert/delete/update of single objects.

5.1 Knowledge in the Knowledge Base

Various kinds of knowledge are needed in order to make the validation process efficient. First, the constraint manager must have knowledge about the way constraints are specified and how they constrain objects in an application. The situation is made complicated by the fact that syntactically different constraints might define semantically equivalent conditions. For example, if we want to isolate managers from employees as a separate relation, the subset constraint between EMP and $MANAGER$ which states that $MANAGER$ is a subset of EMP can be expressed in at least two ways:

$\{m.NAME | m \in MANAGER\} \subseteq \{e.NAME | e \in EMP\}$, or

$(\forall m \in MANAGER)(\exists e \in EMP)(m.NAME = e.NAME)$

Second, the constraint manager should know the types of structures that are available in the database and how objects in the application are represented or mapped into these structures. With such information, it is possible to map certain constraints directly to the database structure and enforce them without extra execution cost. For example, if the database supports network data model, most of the many-to-one cardinality constraints can be implemented as DBTG-set structures.

Finally, the constraint manager must have knowledge about the particular internal structures used to implement these database structures and relate them to objects in the application. The particular file organization and access methods supported by the underlying file system, the cost of each kind of operation, and the current state of the database (e.g., the available indexes, the image sizes of attributes, the sizes of files, and clustering information) should all be taken into consideration to effectively synthesize the constraints.

5.2 Transformation Rules

The constraint manager processes the constraint specifications and synthesizes them into efficient code according to the transformation rules in the knowledge base. A transformation rule is of form "$X : C \rightarrow Y$". X is the expression to be transformed, C is a set of conditions which specifies the class of situations in which the transformation rule can be applied. Y describes the result from applying the rule.

Two categories of transformation rules exist. One category consists of the rules that synthesize the primitive operations into efficient code. Another consists of the rules that transform object specifications by adding internal properties, functions, or data structures. In addition, rules can also be used to derive facts which can be used later in validation, to express knowledge about efficiency characteristics of various usage patterns, or even to govern the order and focus of the rule application process itself.

5.3 Architecture of Constraint Manager

The constraint manager performs two major functions: (1) knowledge acquisition and management, and (2) constraint transformation. It extracts knowledge from the database schema definition, from user-specified constraints, and from the results of continuous monitoring of the database state. Constraint synthesis is done by applying the transformation rules in the knowledge base. The set of rules to be applied depends on the meta-facts available in the knowledge base. The amount of knowledge must be relatively small compared to the database in order to make the constraint validation process efficient. Thus we are not interested in knowledge about individual objects — it is maintained in the database itself. Instead, we are interested in utilizing knowledge about groups of objects, either about some common properties that objects in a group share or about the properties of a group as a whole.

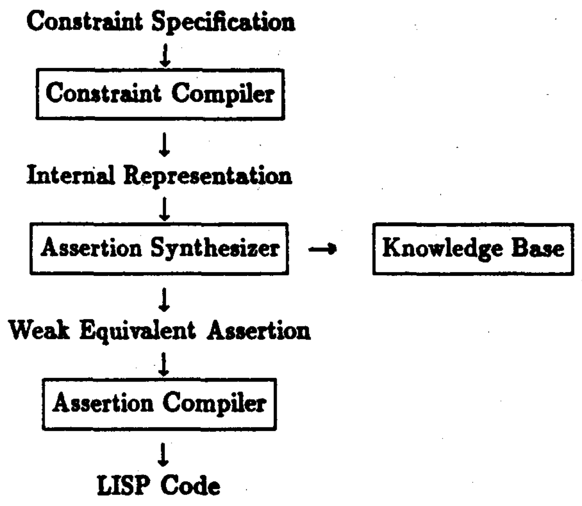

Fig.1 Architecture of Constraint Manager

Three types of actions are possible for the constraint manager to take to validate an integrity constraint. In the simplest situation, the constraint manager does not need to do anything more than checking the update request. The validation of domain constraints is such a case. The constraint manager may have to retrieve further information from the database to validate a constraint. In the most complex case, the constraint manager has to retrieve some data and update some objects in order to obtain another valid database extension. It is the improvement on the last two types of actions that are most beneficial.

The overall organization of the constraint manager is as shown in Fig. 1. It consists of three components. The Constraint Compiler takes as input the user-supplied constraint specifications and compiles them into some internal representations with cost information associated with each of the sub-expressions. The Assertion Synthesizer takes the internal representation of constraints and reformulate them into weak-equivalent assertions when necessary using the knowledge in the knowledge base such that the cost is reduced. Finally the Assertion Compiler compiles these assertions into LISP code which are associated with database manipulation operations.

6 Example Transformation Rules

In this section, we present example transformation rules which cover all types of integrity constraints mentioned before. These rules are intended to be stored in the knowledge base and used by the constraint manager to reformulate constraint assertions into efficient code.

6.1 Domain Constraint Validation

The validation of domain constraint happens whenever an object is INSERTed or an attribute is UPDATEd. The constraint manager evaluates the constraint for each domain involved against the input data and performs the operation only when the constraint is satisfied. No extra information need to be accessed to validate the constraint. The transformation rules for domain constraint validation are:

```
rule DOMAIN-INSERT-VALIDATION
params (P: domain-constraint; D: domain)
transform
  insert(D, OBJECT)
→
  if P(OBJECT) then insert(D, OBJECT)
  else REJECT
```

```
rule DOMAIN-UPDATE-VALIDATION
params (P: domain-constraint; D: domain)
transform
  update(D, OBJECT1, OBJECT2)
→
  if P(OBJECT2) then
    update(D, OBJECT1, OBJECT2)
  else REJECT
```

The propagation of domain integrity constraints through inter-domain functional constraints may make the validation of domain constraints more effective. Because a domain constraint is validated against the input data object, it is most efficient if this validation can be done directly without any transformation on the input object. Such a situation occurs when multiple views are integrated into the database hence many virtual domains are defined as functions of several other domains and the constraints on these domains can be mapped to certain constraints on the virtual domains. Now newly inserted objects of the virtual domains can be validated against the "virtual" constraints.

[Example 9] Let D_1 and D_2 be two domains and f be an inter-domain functional constraint on D_2: $D_2 = f(D_1)$. Assume there is a domain constraint defined on D_1: $(\forall x \in D_1)(P(x))$. A direct validation of operation $INSERT(D_2, d)$ would be to validate the operation $INSERT(D_1, f^{-1}(d))$, i.e., to prove $P(f^{-1}(d))$. A better approach would be to first find out an equivalent domain constraint defined on D_2 : $(\forall x \in D_2)(P'(x))$ such that $(\forall x \in D_2)(P'(x) \equiv P(f^{-1}(x)))$, which will save the evaluation of $f^{-1}(d)$ if the request is invalid. This argument is sound only when f is a one-to-one mapping, i.e., when f^{-1} exists. Although it is possible to propagate domain constraints through many-to-one functional mappings, it usually does not make sense to update a virtual domain which is the result of a many-to-one mapping because such update cannot be mapped to any specific database state transitions.

The transformation rule for domain constraint propagation through an inter-domain one-to-one functional constraint is:

```
rule DOMAIN-CONSTRAINT-PROPAGATION
params (P: domain-constraint; f : $D_1 \mapsto D_2$)
transform
  $P(D_1) : f^{-1} = g \wedge$ one-to-one(f)
→
  $P'(D_2) = P(g(D_2))$
```

6.2 Cardinality Constraint Validation

The important information necessary to validate a cardinality constraint for an inter-domain connection $Conn(R_1 : X_1, ..., R_n : X_n)$ is the number of tuples of each relation involved in the connection. Notice that the size function of a set is continuous with respect to the object-wise changes to the set, finite differencing can be utilized to reduce the cost of validation. In particular, no change can be made, via the operations we consider here, to those domains $R_i, i \in \{1, ..., n\}$ whose $l_i = m_i$. Updates to such relations could only be performed through a transaction mechanism which is beyond the scope of this paper. Counter information can be maintained for those relations whose $l_i < m_i$.

Any operation may invalidate a cardinality constraint and therefore need to be validated before execution. The transformation rules for inter-domain cardinality constraint validation are:

```
rule INTER-DOMAIN-CONNECTION
params ($M : Conn(R_1 : X_1, ..., R_n : X_n)$;
        P:cardinality-constraint)
transform
  $M : l_i < m_i \wedge i \in \{1, ..., n\}$
→
```

```
    $Conn(R_1 : X_1, ..., R_i : X_i : M_i, ..., R_n : X_n) \wedge$
    $VAL = \bigcup_{i=1}^{n} \{v | (\exists r_i \in R_i) \wedge r_i[X_i] = v\} \wedge$
    $(\forall v \in VAL)(M_i(v) = size\{r | r \in R_i \wedge r[X_i] = v\}$

rule INTER-DOMAIN-INSERT
params $(M : Conn(R_1 : X_1, ..., R_n : X_n);$
        P:cardinality-constraint)
transform
    $insert(R_i, r) : l_i < m_i \wedge i \in \{1, ..., n\}$
$\longrightarrow$
    if $M_i(r[X_i]) < m_i$ then
        $M_i(r[X_i]) \leftarrow M_i(r[X_i]) + 1$
        $insert(R_i, r)$
    else REJECT
```

6.3 Value Constraint Validation

Inter-domain value constraint has the general form $P(f_1, ..., f_m)$ where $f_i, i = 1, ..., m$ are functions on domains $D_i, i = 1, ..., n$ and P is a predicate on these function values. In order to evaluate P efficiently, information may be maintained about the current values of these functions. The transformation rules in the knowledge base can be applied to determine exactly what kind of auxiliary information are needed and how to synthesize the database operations to maintain such information.

The general strategy for inter-domain value constraint validation can be stated as the following steps: (1) Determine, for all $f_i, i = 1, ..., m$, whether f_i is continuous with respect to its parameters, i.e., to compute the set:

$$S = \{(f_i, p_{ij}) | 1 \leq i \leq m \wedge 1 \leq j \leq l_i \wedge continue(f_i, p_{ij})\}$$

where l_i is the number of parameters of f_i, (2) Apply the rules in the knowledge base to remove discontinuity as much as possible, (3) Apply transformation rules to set up the auxiliary data structures, functions, etc. and synthesize operations, and (4) Whenever a change is made to a discontinuous parameter of a function, recompute all the relevant auxiliary information.

[Example 10] Transformation rules for single operator "*max*" in function $f = max(S)$ where S is a set of elements on which *max* is meaningful.

```
rule MAX-FUNCTION
params($S$ : set; $f$ : function)
transform
    $f : element \in S \wedge f : S \mapsto element$
$\longrightarrow$
    $f : \sharp MAX$

rule MAX-INSERT
params($S$ : set; $f$ : function)
transform
    $insert(S, s) : f = max(S) \wedge f : S \mapsto element$
$\longrightarrow$
    $insert(S, s)$
    if $\sharp MAX < s$ then $\sharp MAX \leftarrow s$

rule MAX-DELETE
params($S$ : set; $f : S \mapsto element$)
transform
    $delete(S, s) : f = max(S)$
$\longrightarrow$
    $delete(S, s)$
    $\sharp MAX \leftarrow s' \in S$
    foreach $s'' \in S$ do
        if $s'' > \sharp MAX$ then $\sharp MAX \leftarrow s''$
```

[Example 11] Transformation rules for a function f over sets $S_i, i = 1, ..., n$ which is a composition of two operators *max* and $\cup$: $f = max(\bigcup_{i=1}^{n} S_i)$.

```
rule MAX-OVER-UNION
params($f$ : function; $n$ : integer; $S_1, ..., S_n$ : set)
transform
    $f : f = max(\bigcup_{i=1}^{n} S_i) \wedge n > 1$
$\longrightarrow$
    $f : \sharp MAX$

rule MAX-OVER-UNION-INSERT
params($f$ : function; $n$ : integer; $S_1, ..., S_n$ : set)
transform
    $insert(S_i, s) : 1 \leq i \leq n \wedge f = max(\bigcup_{i=1}^{n} S_i)$
$\longrightarrow$
    $insert(S_i, s)$
    if $\sharp MAX < s$ then $\sharp MAX \leftarrow s$
```

[Example 12] Transformation rules for discontinuity removal of *avg* in function $f = avg(S)$.

```
rule AVG-CONTINUE
params($f$ : function; $S$ : set)
transform
$f : f = avg(S)$
$\longrightarrow$
$sum(S)/size(S) : \sharp SUM, \sharp SIZE$
```

7 Conclusion and Future Work

Experience has shown that the conventional approach to integrity constraint enforcement is not successful even for simple static constraints and single update operations. Because of the very nature of integrity constraints, the richness of their specification in first order logic, and the essential demand for the efficient validation, a knowledge-based approach seems to be the most promising direction. In this paper, we have demonstrated the feasibility and power of such an approach to the validation problem of static integrity constraint with repect to single update requests. It is our belief that the use of AI techniques will be very fruitful in this problem domain.

Clearly, this is only a proposal and much more work needs to be done to prove the concepts. We plan to implement a primitive set of transformation rules in the environment of KSYS, which is a frame-based knowledge base system, and test our ideas using a specific database of ships and ports. Rules for efficiency estimators, rule applications, functional operators, function composition, etc. need to be refined and the techniques for integrity constraint validation in the transaction environment need to be developed.

8 Acknowledgement

The authors are grateful to the colleagues in the KSYS group, especially Waqar Hasan, for helpful discussions. Suggestions from Prof. Shixuan Sa of the People's University of China are also appreciated. This work was supported by DARPA contract N39-84-C-211 for Knowledge Based Management Systems.

9 References

[1] Badal, D. and Popek, G., "Cost performance analysis of semantic integrity validation methods;" *Proc. ACM SIGMOD*, 1979, 109-115.

[2] Bernstein, P., Blaustein, B., and Clarke, E., "Fast maintenance of semantic integrity assertions using redundant aggregate data;" *Proc.6th Int.Conf. VLDB*, 1980, 126-136.

[3] Brodie, M., "Specification and Verification of Database Semantic Integrity;" PhD. Dissertation, *Tech. Report* CSRG-91, Univ. Toronto, April 1978.

[4] El-Masri, R. and Wiederhold, G., "Properties of Relationships and their Representation;" *Proc. of the 1979 NCC*, AFIPS vol.49, Aug. 1979, 319-326.

[5] Furtado, A., dos Santos, D., and de Castilho, J., "Dynamic modelling of a simple existence constraint;" *Inf. Syst.* 6, 1981, 73-80.

[6] Goldberg, A. and Kotik, G., "Knowledge-Based Programming: An Overview of Data Structure Selection and Control Structure Refinement;" Kestrel Inst./Univ. of CA, Santa Cruz report, Kes.U.83.7, 1983.

[7] Green, C. and Westfold, S., "Knowledge-based programming self-applied;" *Artificial Intelligence 10*, Ellis Forward & Halsted Press (John Wiley), 1982.

[8] Hammer, M. and McLeod, D., "Semantic Integrity in a Relational Database;" *Proc.1st Int.Conf. VLDB*, Framingham, Mass., 1975.

[9] Hammer, M. and McLeod, D., "A Framework for Database Semantic Integrity;" *Proc. 2nd Int.Conf. on Software Engineering*, San Francisco, 1976.

[10] Keller, A.M. and Wiederhold, G., "Validation of Updates Against the Structural Database Model;" *Proc. of Symposium on Releability in Distributed Software and Database Systems*, Pittsburgh, July 1981.

[11] Kotik, G., "Knowledge-based compilation of high-level data types;" Kestrel Inst./Univ. of CA, Santa Cruz report, 1983.

[12] Morgenstern, M., "The Role of Constraints in Databases, Expert Systems, and Knowledge Representation;" *Proc.1st workshop on Expert Database Systems*, Oct. 1984.

[13] Paige, R., "Applications of finite differencing to database integrity control and query/transaction optimization;" *Advances in database theory, Vol.2*, ed. H. Gallaire, J. Minker and J. Nicolas, Plenum Press, New York.

[14] Shepherd,A. and Kerschberg,L., "PRISM:A Knowledge-Based System for Semantic Integrity Specification and Enforcement in Database System;" *Proc.ACM SIGMOD Conf.*, Boston, 1984, 140-173.

[15] Shepherd, A. and Kerschberg, L., "Constraint Management in Expert Database Systems;" *Proc.1st workshop on Expert Database Systems*, Oct. 1984.

[16] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification;" *Proc. of the 1975 SIGMOD Conference*, ACM SIGMOD, San Jose, June 1975.

[17] Tsichritzis, D. and Lochovsky, F., *Data Models*; Prentice-Hall, NJ, 1982.

[18] Wiederhold,G. and El-Masri,R., "Structural Model for Database Systems;" Stanford University, Computer Science Department report CS-79-722, April 1979.

[19] Wiederhold,G., "Knowledge and Database Management;" *IEEE Software* Premier Issue, vol.1 no.1, Jan. 1984, 63-73.

[20] Wiederhold, G., "Knowledge versus Data;" to appear in *On Knowledge Base Management Systems: Integrating AI and Database Technologies*, ed. M. Brodie, 1986.