# A Syntactic Method for Enforcing Semantic Database Integrity Constraints

Robert R. Goldberg [1], Jacob Shapiro [2], Jerry Waxman [1]

[1]Department of Computer Science, Queens College of CUNY, Flushing, USA

[2]Department of Statistics and Computer Information Systems, Baruch College of CUNY, New York, USA

An automata theoretic framework is proposed which allows for a syntactic treatment of semantic properties of relational databases.The semantic integrity of a database is defined in terms of predicates on its attributes. These predicates are shown to determine the specification of a finite state automata which detects, in time proportional to the length of the input, whether the semantic integrity of the database has been violated. In addition, this approach leads to efficient techniques for dealing with the problems of consistency, equivalence and redundancy of integrity constraints. These concepts are defined and algorithms for their determination are described.

## 1. Introduction

An area of primary concern in database systems design is that of maintaining the integrity of the data stored in the database. The integrity problem has many facets ranging from data entry verification to the maintenance of concurrently usable files [9]. One of the most important concerns in data integrity is that of maintaining semantic integrity of database records.

The term semantic integrity is generally taken to mean that the principles or rules underlying the relationships between the data items in the database are not violated. Figure 1 illustrates part of a typical database record containing employee information.

Among the various semantic relations that might obtain are:

1. Emp-Sal < Man-Sal
2. Man-Sal < $75,000
3. If Emp-No. > 1000 Then
   Emp-Sal < $35,000

A slightly more complex example might be the following :

If Field_10 contains code "a" and Field_150 contains a "b", then Field_225 may only contain entries from the set {0,1,2}.

Database records can often contain hundreds of fields. The list of integrity constraints can be very long, and their complexity can be quite great.

Semantic integrity may be applied on many levels in a database system. Constraints may apply to individual fields or between the fields of a record. There may be intra-record constraints or constraints that apply globally to the set of all database records in a given data file. In addition, there may be constraints connecting records across the tables comprising the complete database system.

| EMP-NAME | EMP-NO | EMP-SAL | EMP-ID | MAN-SAL |
|---|---|---|---|---|
| Employee's Name | Employee's Number | Employee's Salary | Manager's ID | Manager's Salary |

Figure 1: A Database Record Containing Some Employee Information

This paper will deal with the first of these levels: enforcing constraints on individual fields or between the fields of a record. We provide a method for automatically transforming potentially complex constraints between the fields of a record to a finite state automata which will enforce the semantic integrity based on the syntactic structure of incoming records (transactions) that are used to update the database. The focus of the method will be to provide a real-time filter to guarantee that data passed to the database conforms to proscribed semantic constraints.

Aside from automatic real-time constraint enforcement, this approach has additional benefits as well. In particular, it will allow for

1) automatic constraint consistency checking,
2) the ability to test for the equivalence of alternate constraint formulations, and
3) automatic constraint redundancy checking.

Typically, constraints are defined on and between the fields by a series of relational equations. One obvious desideratum for constraint specification is that of constraint consistency; that is, the set of constraints should not be mutually contradictory. However, when the number of fields is large, the relationships between them can be quite complex. The standard approach to constraint satisfaction checking is to simply translate the constraint set into a sequence of "if-then-else" statements in a program [9,11]. If the set of constraints is complicated it is quite likely that inconsistencies, if they exist, can be overlooked. The approach presented in this paper will allow for the automatic detection of inconsistent constraints.

Under current practice, redundant constraint specification is not easy to detect. Redundant constraints present two problems in database systems. First, they degrade program efficiency because equivalent conditions are being checked multiple times. Second, they may severely compromise integrity checking, in general. If a set of constraints is, in fact, redundant, a situation occurs in which some conditionmay be derived in more than one way. Changing some of the conditions could leave the constraint set either inconsistent or produce a situation with undesirable implications. This becomes all the more problematic when constraints can change dynamically as is the case with modern database systems.

In addition, constraint sets which look quite different might in fact define the same set of update records for a given database. The ability to test for equality of different formulations of constraints for a given record might be important in some contexts. The language-automata theoretic framework proposed in this paper provides a simple and effective method for testing equivalence of sets of constraints.

In the next section we introduce a predicate language defined over database attributes. Section 3 reviews regular sets and finite state automata (f.s.a.) and section 4 shows how the predicate language is related to f.s.a. In addition, we show how these automata can be used to enforce the integrity constraints imposed by the predicates. In section 5, the notions of consistency, equivalence and redundancy of a constraint set are framed in automata-theoretic terms and algorithms for detecting these conditions are described.

## 2. Predicates over Database Attributes

We start with a brief review of the relevant concepts from database theory, and describe the notation which will be used below. For a full treatment of the appropriate notions see, for example, Ullman [15] or Elmasri et al [6].

A relation schema R denoted by $R(A_1, A_2, \ldots, A_n)$ is a set of attributes $R = \{A_1, \ldots, A_n\}$. Each attribute $A_i$ is the name used to denote some domain D in the relation schema R. D is called the domain of $A_i$ and is denoted by $dom(A_i)$. A relation instance of the relation schema $R(A_1, A_2, \ldots, A_n)$, denoted by r(R), is a set of n-tuples $r = \{t_1, t_2, \ldots, t_m\}$. Each n-tuple t is an ordered list of n values $t = <1, v_2, \ldots, v_n>$, where each value $v_i$, $1 \le i \le n$, is an element of $dom(A_i)$, or is a special value *null*. Notice that r(R) is a subset of the cartesian product of the domains that define R :

$$r(R) \subseteq dom(A_1) \times dom(A_2) \times \ldots \times dom(A_n).$$

For simplicity of notation we will often refer to $dom(A_i)$ by $A_i$, and to r(R) by R. From the context of the discussion it will be clear when we refer to the name of the attribute $A_i$, or to the domain of $A_i$, $dom(A_i)$. Similarly, this will be apparent with r(R) and R. For example, when we say cardinality of $A_i$, we mean $|dom(A_i)|$.

The set of predicates which will be used to define the semantic integrity constraints for a given tuple may be defined formally as follows :

constants: $A_i$ , ... , $A_n$ - symbols corresponding to attributes of R;

$K_1$ , ... , $K_m$ - symbols representing all possible values of elements in $\bigcup_{i=1}^{n} dom(A_i)$;

*null* - symbol indicating that the value in the attribute is omitted.

binary predicates: , =

logical predicates: V ('or'), ¬ ('not').

A **basic predicate** is of the following form: $X\alpha Y$ where

$\alpha$ is one of the binary predicates above;

$X = A_i$ for some $1 \le i \le n$;

$Y = A_j$ for some $1 \le j \le n$ or, $Y = K_l$ where $K_l$ is a constant corresponding to a value in $dom(A_i)$.

A **general predicate** in the language is defined recursively as follows:

1. A basic predicate is a predicate;
2. If P is a predicate then ¬P is a predicate;
3. If $P_1$, $P_2$ are two predicates then $P_1 \vee P_2$ is a predicate.

This recursion will only be applied a finite amount of times.

Note that the predicates as just defined allow for very natural and important integrity constraints to be formulated.

## Examples:

| **Integrity Constraint** | Predicate Statement |
|---|---|
| 1. The values of attribute $A_i$ are limited to the values in the set $\{K_1, K_2 \ldots, K_m\}$. | $\bigvee_{j=1}^{m} (A_i = K_j)$ |
| 2. The values of attributes $A_i$ are between 100 and 200 (i.e. $100 \le A_i \le 200$) | $((A_i < 200) \vee (A_i = 200)) \wedge \neg(A_i < 100)$ |
| 3. If attribute $A_1$ has value $K_1$ then ($A_2$ is either equal to $K_2$ or to $K_3$) and $A_3$ is *null* | $(A_1 = K_1 \rightarrow ([(A_2 = K_2) \vee (A_2 = K_3)] \wedge A_3 = null$ |

Notice that in the predicate statements we used symbols ∧ and →. These are just shorthand and are represented as $A \wedge B = \neg(\neg A \vee \neg B)$, $A \rightarrow B = \neg A \vee B$. One may, of course, introduce higher level predicates defined in terms of basic predicates. For example, one may wish to introduce the predicate "element of" denoted by "∈" to correspond to the predicate of example 1 above. All such definitions, however, must be formulated in terms of some set of elementary predicates and should be considered as a "macro" notation for them.

The set of predicates defined above can require considerable cost for integrity checking. A database administrator or programmer may specify predicates of arbitrary complexity in the number of terms, and, while a preprocessor might be employed to convert the predicates into either disjunctive or conjunctive normal forms, the cost of checking the predicate may grow exponentially in the number of terms. Clever rearranging of the terms for testing a composite predicate might result in faster integrity checking. However, when the predicates contain many terms and refer to many fields, this optimal rearrangement may be far from obvious.

An alternate formulation of predicates as describing structured rules for the formation of strings of a type 3 regular language is now presented. The advantage of this formulation is that it will allow for the automatic generation of a validity checking scheme that will operate in time proportional to the number of attributes.

## 3. Regular Sets and Finite State Automata

For the sake of completeness, the reader is reminded of basic definitions and results from formal language and automata theory which are important in formulating our method for integrity checking. For a detailed treatment see Hopcroft and Ullman [10]. First we define the concept of a regular set.

**Definition :**

A **regular set** over an alphabet Σ is a set of strings constructed from Σ as follows :

Base Case : ∅, {λ}, Σ are regular sets.

Recursive Step :

a) If $A_1$, $A_2$ are regular sets, then $A_1 \cup A_2$ is regular.

b) If $A_1$, $A_2$ are regular sets, then $A_1 \cdot A_2$ is regular where $A_1 \cdot A_2 = \{uv \mid u \in A_1 \& v \in A_2\}$.

$A^{[i]}$ means A concatenated with itself $i$ times. $A^0 = \{\lambda\}$; $A^1 = A$.

c) If A is regular, then A* is regular. The Kleene Star operator, *, is defined in terms of union and concatenation by $A^* = \bigcup_{i=0}^{\infty} A^{[i]}$.

Closure : The recursion step may be applied any finite number of times [14].

**Lemma**. Every finite set of strings is regular.

**Proof**. Each individual string may be formed by the concatenation of its underlying individual symbols. A string is then represented by the set that contains it, and hence is regular. Therefore, a finite set of strings may be formed by the union of the sets of the individual strings, and is therefore also regular. □

Regular sets may be used to naturally express the set of tuples that forms the valid insertions to a database. The cartesian product of the attributes $\underset{i}{x} A_i$ may be viewed as a set of all concatenations of the form $a_1a_2...a_n$ where $a_i \in A_i$, which may be ordered without loss of generality. Therefore, every tuple can be represented by an unique string. Also notice that each attribute $A_i$ is a finite set of strings and hence is regular. Therefore, $\underset{i}{x} A_i$ is also regular. In particular, any subset of $\underset{i}{x} A_i$ is finite and by the previous lemma, any set of relations is also regular.

**Definition :**

A **finite state automata** is a quintuple $M = (\Sigma, Q, \delta, S_0, F)$ where $\Sigma$ is a finite set of input symbols, Q is a set of states, $\delta: \Sigma \times Q \rightarrow Q$ is the next state function, $S_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of final states.

A finite-state automata is a mathematical abstraction of "machines" or processes and is characterized by the following three properties:

(1) The automata may be said to be in one of a finite number of states at a particular time. Some subset (usually non-empty) of these states is denoted as the set of final states.

(2) The state of the automata at time $t_{i+1}$ is a function of just the state of and the input to the automata at time $t_i$.

(3) Some subset of the set of states is called the final state set, and if, when the machine has processed the last input, it transfers to a final state, it is said to have "accepted" the input string.

A classic result, Kleene's Theorem, relates regular sets with finite-state automata (f.s.a.). By Kleene's Theorem, for each regular set L, there exists a finite-state automata $M_L$ such that the set of strings for which $M_L$ enters a final state is precisely L. Moreover, the standard proof of this correspondence is constructive; that is, an algorithm exists that, when given a generator for a regular set L will produce a finite-state automata accepting L [10].

Kleene's Theorem has important implications of the above discussion to the problem of constraint enforcement. We will demonstrate below how to construct a machine M simulating each ***P***, the set of integrity constraints. The acceptance of a string by M will indicate that the associated tuple is valid.

## 4. Predicates and Finite State Automata

Consider a relation R over n attributes, $R(A_1, A_2, ..., A_n)$. We assume that R is a finite set of tuples. Let the integrity constraints defined on R be given by the finite set of predicates $\boldsymbol{P} = \{P_1 ... P_M\}$. That subset of R satisfying ***P*** is, of course, also finite. Hence by the above discussion, the set of valid tuples in R constitutes a regular language.

It should be noted that the standard proof of the fact that a finite set is accepted by some f.s.a. involves a construction which essentially assigns a list of states to each element of an accepted string in the set [5]. If the set is large, this process is extremely impractical.

We will now present an alternative scheme which will allow for the machine description to be read off directly from the predicate. We will first show how to associate a machine with each basic predicate and then how to construct a machine for general predicates.

## Basic Predicates

**Case 1:**

P is a predicate of the form

P: $A_i \alpha K$

This denotes that subset of R whose $i^{th}$ entry $A_i$ stands in relation $\alpha$ to the constant K.

Let $UA = \bigcup_{i=1}^{n} A_i$. Consider a f.s.a. $M = (\Sigma, Q, \delta, A_i, T_s)$ where $\Sigma = UA$ and $Q = (A_1', A_2' \ldots \ldots A_i', T_F, T_S)$.

Note that the input set is the union of all the valid attribute values names up to and including those from $A_i$. There are two additional states $T_S$ and $T_F$ which are used to indicate success ($T_S$) and failure ($T_F$).

Let $\alpha$ (K) = {x | x $\in A_i$ and x $\alpha$ K}. Let $a$, $b_\alpha$ and $c_{\bar{\alpha}}$ denote generic elements of UA, $\alpha$(K), and UA-$\alpha$(K) respectively.

We may define machine M as follows:

1. $\delta(A_j', a) = A_{j+1}', 1 \le j \le i$
2. $\delta(A_i', b_\alpha) = T_S$
3. $\delta(A_i', c_{\bar{\alpha}}) = T_F$
4. $\delta(T_S, a) = T_S$
5. $\delta(T_F, a) = T_F$

It is easy to show that M will accept precisely that subset of R for which $A_i \alpha K$. Since the only critical element for ***P*** is the value of attribute $A_i$, when the $i^{th}$ input is encountered, if it is an element of $\alpha$ (K), M enters state $T_S$, which is a final state (by (2)) and stays there (by (4)). If the value of $A_i$ is $c_{\bar{\alpha}} \notin \alpha$ (K), then M enters state $T_F$ (by (3)) and stays there (by (5)).

Notice that this description of M is not in the standard form defined in the previous section. The basic difference lies with the elements $a$, $b_\alpha$ and $c_{\bar{\alpha}}$, since they represent generic elements of their respective sets. The definition of $\delta$ is also not in standard form. Each rule should be replaced by a **set** of rules, one for each element represented by the respective symbol $a$, $b_\alpha$ and $c_{\bar{\alpha}}$. Rule (1), for example, would be replace by |UA| rules.

In a practical algorithm this rewriting need not be done, since we need not check the input symbol except for the $i^{th}$ input. Thus, we may effectively check the predicate P: $A_i \alpha K$. The time required to test a given element of R for membership in $R_P$, for this P, is therefore proportional to $n$, since each element in attribute $A_j$ for $j \neq i$ will just be scanned and $A_i$ will be evaluated by a function call that takes some bounded time.

In addition, the rules as written would not require that each $R_j$ be checked to see if the input value were an element of $A_j$. This is not required by the predicate $A_i \alpha K$ and the rules allow any symbol in fields other than $A_i$. In practice, however, the list of predicates defining the integrity of a given record type would include predicates to guarantee that the $j^{th}$ input is of a type valid for the $j^{th}$ field.

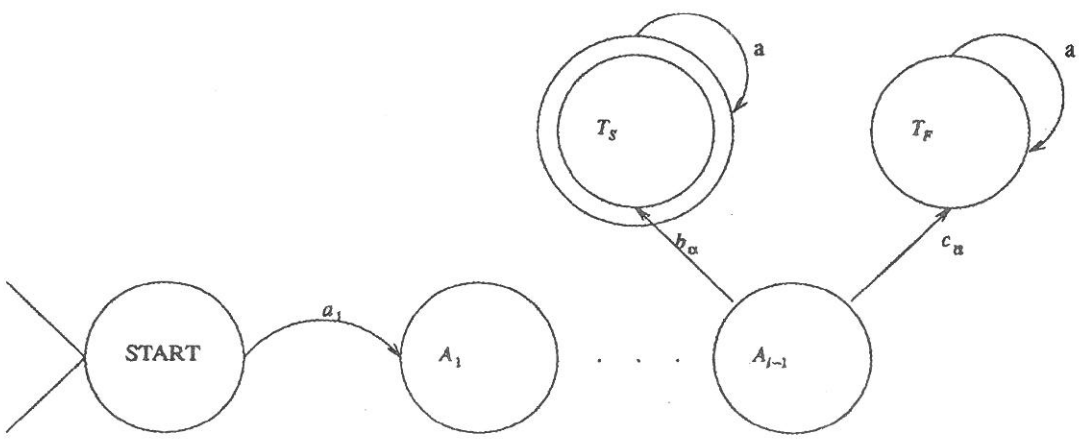



Figure 2: Finite State Automata for Single Predicate of the Form P: $A_i \alpha K$.

**Case 2:**

P is a predicate of the form

P: $A_i \; \alpha \; A_j$

Assume, without loss of generality, that i<j. For each x $\in A_i$, let $\alpha(x)$ denote that subset of $A_j$ whose elements stand in relation $\alpha$ to x. Then, $\alpha(x) = \{y | y \in A_j \text{ and } x \alpha y\}$. The key to creating a machine for the predicate term P: $A_i \; \alpha \; A_j$ is to establish, after the $i^{th}$ input, an $|A_i|$-way branch. Each of the branches will be testing for a particular $x_L \in A_i$ and so the results of the previous discussion apply. As above let $a$ «UA; $x_L \in A_i$ be defined as above.

Let machine M be as follows:

$M = (\Sigma, Q, \delta, A^{/}, T_S)$

where:

$\Sigma = UA$

$Q = (A^{/}_1, A^{/}_2 \ldots A^{/}_i, A^{/}_{(L,M)}, T_S, T_F)$, $1 \le L \le |A_i|$, $i \le M \le j-1$

$\delta$:

1. $\delta(A^{/}_M, a) = A^{/}_{M+1}$; $1 \le M \le i-2$
2. $\delta(A^{/}_{i-1}, x_L) = A^{/}_{(L,i)}$; $1 \le L \le |A_i|$
3. $\delta(A^{/}_{(L,M)}, a) = A^{/}_{(L,M+1)}$; $1 \le M \le j-1$, $1 \le L \le |A_i|$
4. $\delta(A^{/}_{(L,j-i)}, y) = T_S$, $y \in \alpha(x_L)$; $1 \le L \le |A_i|$
5. $\delta(A^{/}_{(L,j-i)}, y) = T_F$, $y \in \alpha(x_L)$; $1 \le L \le |A_i|$
6. $\delta(T_F, a) = T_S$

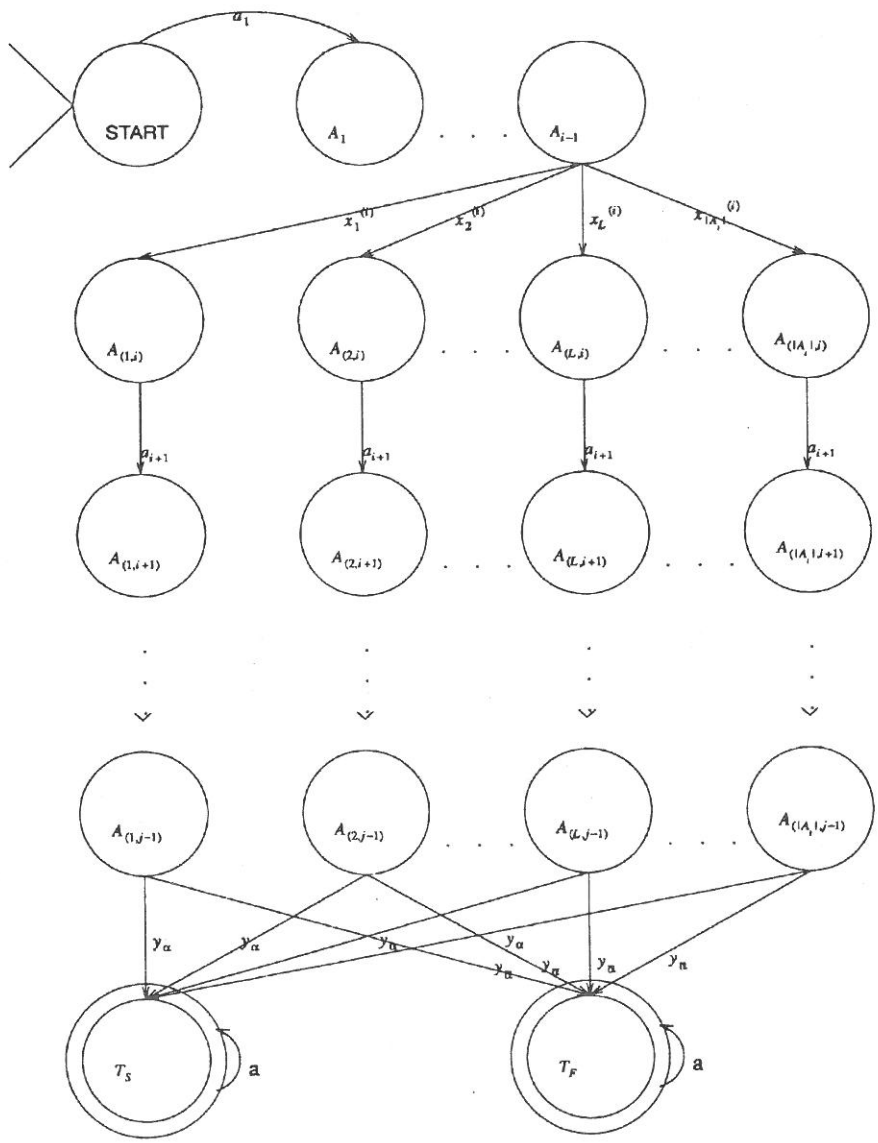



Figure 3: Finite State Automata for Predicate Form P: $A_i \; \alpha \; A_j$ .

As defined, **M** will accept all and only those tuples for which $A_i \alpha A_j$. In the set **Q**, $A'_{(L,M)}$ generically represents a set of $|A_i| * (j-i)$ states. Specifically, there are $|A_i|$ states of the form $A'_{(L,1)}$ each corresponding to a particular $x_L \in A_i$. The chain of states leading from $A'_{(L,1)}$ to $A'_{(L,j-1)}$ represents a copy of the machine which accepts strings satisfying the predicate $A_j \alpha x_L$. Consider a tuple in R for which $A_i \alpha A_j$ is satisfied. The first i-1 entries of the tuple cause the machine to go from $A'_1$ to $A'_i$ (by rule 1). At $A_i$ and depending on $x_L \in A_i$, **M** enters $A'_{(L,1)}$ (by rule 2) and further inputs cause it to transverse the states $A'_{(L,2)} \ldots \ldots A'_{(L,j-1)}$ (by rule 3). At $A'_{(L,j-i)}$ any y such that $y \in \alpha(x_L)$ will cause **M** to enter $T_S$ (by rule 4). Therefore, if $A_i \alpha A_j$ is satisfied, **M** enters a final state and cycles there (by rule 6). If, however, $A_i \alpha A_j$ is not satisfied, then at $A'_{(L,j-i)}$, y will not be an element of $\alpha(x_L)$ and hence **M** will enter $T_F$ (by rule 5). $T_F$ is a trap state (by rule 7) and hence **M** halts and the tuple is rejected.

For predicate terms of the form $A_i \alpha K$, the number of nodes needed for machine **M** is i+2. For predicate terms of the form $A_i \alpha A_j$, the number of nodes needed for machine **M** is $i+(j-i)*|A_i|+2$ and the number of input symbols is, as before, $|UA|$. Likewise, this number may be reduced and still maintain the linearity of the integrity checking by introducing generic elements representing certain classes of inputs in a way analogous to the machine for $A_i \alpha K$ (figure 2); we let *a* represent any symbol in UA, and *b* and *c* represent generic symbols in $\alpha(x_i)$ and $UA-\alpha(x_i)$ respectively for $1 \le i \le |A_i|$. This represents in practice a significant decrease in size as compared with UA.

Generic elements allow for the input domain of each attribute to be handled efficiently. Consider some large set $A_i$. If a transition were stored in the state able for each element, the size of this table could be prohibitive, although clearly the search time for the transition would be constant. In many applications, the domains of each attribute are quite tractable. The reason for this is that there are many application areas in which the domain is in fact small; moreover, even in those instances when the domain appears to be large (for example in a SALARY domain), many times the items of interest are ranges of that domain. For example, consider when the SALARY is under 10,000, between 10,000 and 50,000, and, over 50,000. Also, notice that as the number of generic elements, or categories rises, the complexity for obtaining the correct transition between one state and the next increases. In fact, this is directly related to the number of constraints. However, as the linear record of length n is input, only n transitions are necessary to get from the start to the final (and possibly accepting) state. If the size of the domain of the attribute is tractable, the time to ascertain the subsequent state is constant, based on the initial construction of the transition matrix. If not, then in worse case a tree search of subsets of the resultant attribute domains (after the constraints have been implemented and the machine minimized [10]) would be necessary to ascertain the state that the current transition should take. This however would only add a log(constraints) factor to each of the transitions that have large number of such domain subsets. We are still better off than with a possible exponential number of states.

## General Predicates

We have demonstrated how individual predicate terms defining an integrity constraint for a given relation schema may be converted to the description of a f.s.a. which accepts only those strings from $A_1 x \ldots x A_n$ satisfying the particular term. What about combinations of such terms? While a predicate may be an arbitrarily complex combination of terms, the construction of a machine for accepting them is not conceptually difficult since the only connectives that may be used in constructing predicates from terms are boolean connectives. The proof of the following theorem shows how this may be done.

**Theorem.** Any Boolean combination of predicate terms determines a regular language.

**Proof.** Formally, let $\alpha_1$ and $\alpha_2$ be predicate terms on schema R. Since both $\alpha_1$ and $\alpha_2$ determine finite subsets of tuples from r(R), and hence are regular, by definition the union of them also is a regular language. Moreover, algorithmic techniques exist which, when given machines corresponding to $\alpha_1$ and $\alpha_2$, will produce a machine recognizing $\alpha_1 \vee \alpha_2$, $\alpha_1 \wedge \alpha_2$ and $\overline{\alpha}$ [14]. The ideas involved are straightforward. If $M_{\alpha_1}$ and $M_{\alpha_2}$ are machines corresponding to $\alpha_1$ and $\alpha_2$ respectively, then $L(M_{\alpha_1}) \cup L(M_{\alpha_2})$ may be recognized by a non-deterministic finite-state machine, by ad-

ding a new starting state with nondeterministic transitions to the starting states of $M_{\alpha_1}$ and $M_{\alpha_2}$ (figure 4). If $M_\alpha$ accepts all tuples corresponding

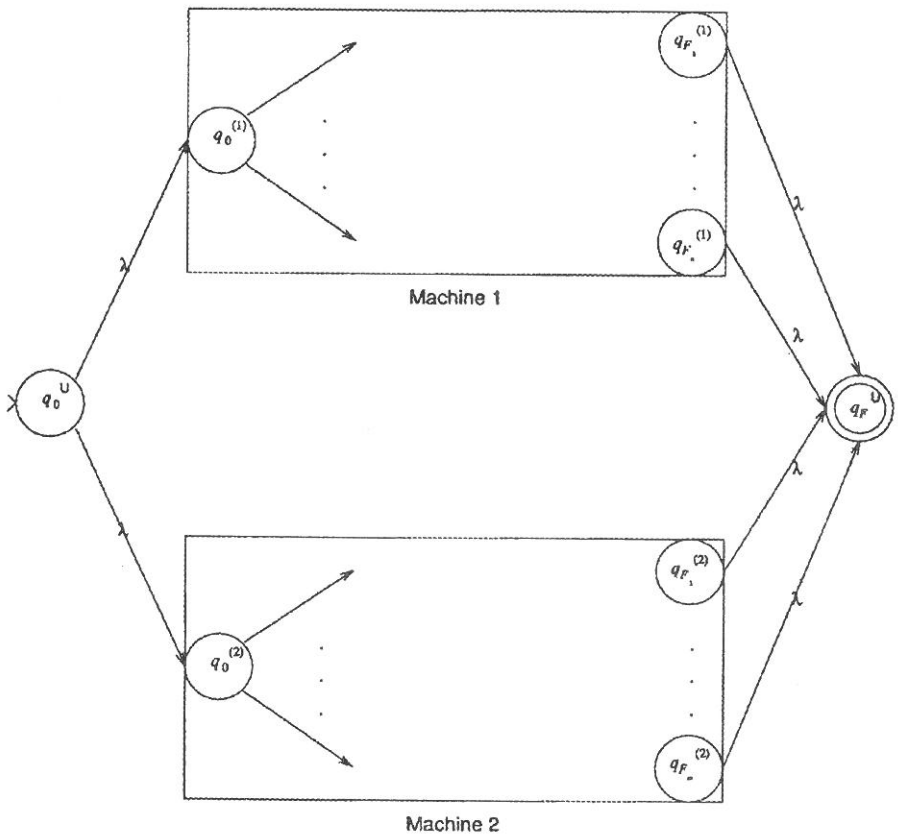
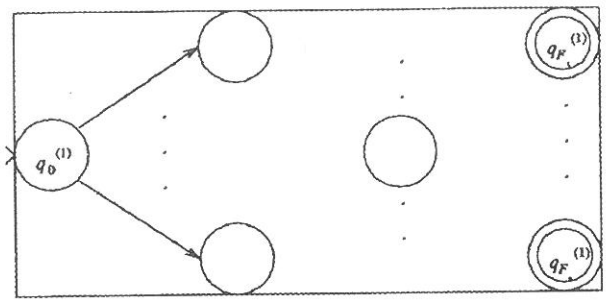



Figure 4 : Joining two finite state machines with a $\lambda$. This is accomplished by adding nondeterministic ($\lambda$, free) transitions between nodes.

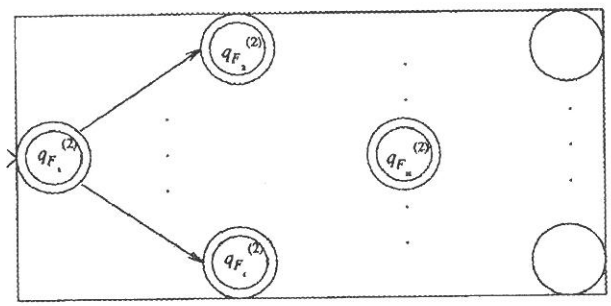
to the predicate $\alpha$ then $M_{\overline{\alpha}}$, which is equivalent to $M_\alpha$ except with $T_S$ and $T_F$ interchanged, will accept $\overline{\alpha}$ (figure 5). Finally, since $\alpha_1 \cap \alpha_2$ is equivalent to $\overline{\overline{\alpha_1} \cup \overline{\alpha_2}}$, and $\alpha_1 \rightarrow \alpha_2$ is equivalent to $\neg\alpha_1 \vee \alpha_2$, the previous results apply and their machine representations may be constructed as above.

□

If $M_1$ and $M_2$ are machines for predicates $\alpha_1$ and $\alpha_2$ respectively, then a machine for $\alpha_1 \vee \alpha_2$ requires at most $|Q_{M_1}| + |Q_{M_2}| - 2$ states since only one trap and final state are needed. If $M_1$ recognizes $\alpha_1$, then $M_2$ which recognizes $\overline{\alpha}_1$, requires $|Q_{M_1}|$ states, since only one interchange of the trap and final states is needed. For the predicate $\alpha_1 \cap \alpha_2$, one needs at most $|Q_{M_1}| + |Q_{M_2}| - 2$ states as well. Since the essential cost will be in the union of machines to recognize $\alpha_1$ and $\alpha_2$, each negation does not add states and neither does the final one. Predicates of arbitrary complexity may thus be implemented and their cost of implementation as measured by the number of states necessary for their construction may be iteratively obtained. The machine thus obtained is in fact nondeterministic. Standard approaches for transforming to a minimal deterministic finite state machines could in worst case produce an intermediate machine of high memory requirements. However, this would only be applicable if the conglomerate nondeterministic machine was minimized at the end of joining together all of the constraints. The approach presented here iteratively obtains a minimal deterministic finite state machine, as each constraint is added. Consequently, no exponential costs of intermediate machine constructions are incurred during the minimization process.



Machine 1



Machine 2

Figure 5 : The language of machine 2 is the complement of that the language of machine 1. This is accomplished by simply switching the designation of success (final) and failure states.

## 5. Consistency, Equality, and Redundancy

The automata theoretic perspective allows for the simple derivation of consistency, equality and redundancy of constraint sets. We start with the definition of satisfaction.

**Definition:**

Let $t = \langle v_1, \ldots, v_n \rangle$ be a tuple in R where $v_i \in dom(A_i)$, $1 \le i \le n$. Let P be a basic predicate; then P has one of the following four forms:

1) $A_i = A_j$;

2) $A_i < A_j$;

3) $A_i < K$;

4) $A_i = K$.

We say that t **satisfies** P if for each of the forms above the following holds respectively:

1) $v_i = v_j$;

2) $v_i < v_j$;

3) $v_i < K$;

4) $v_i = K$, where $K \in dom(A_i)$ corresponding to a specific constant K.

Satisfaction for general predicates is defined in the obvious way. If t satisfies P then t does not satisfy $\neg P$.

If t satisfies at least one of $P_1$ , $P_2$, then t satisfies $P_1 \vee P_2$.

The above definition may be generalized to a set of predicates in the following way :

**Definition:**

Given a relation R, and a set $\boldsymbol{P} = \{ P_1 , \ldots , P_m \}$ of predicates we say that R **satisfies** $\boldsymbol{P}$ if for every tuple t in R and every $P_i$ in $\boldsymbol{P}$, t satisfies $P_i$.

We can now define consistency, equivalency and redundancy.

**Definition:**

A set $\boldsymbol{P}$ of predicates is **consistent** if there exists a relation R such that R satisfies $\boldsymbol{P}$.

Two sets of predicates $P_1$ and $P_2$ are **equivalent** if for every R, R satisfies $P_1$ *iff* R satisfies $P_2$.

A set of predicates $\boldsymbol{P}$ is **redundant** if there exists a proper subset $P_1$ of $\boldsymbol{P}$ such that $P_1$ equivalent to $\boldsymbol{P}$.

The next two theorem shows how the consistency of a set of predicates may be determined.

**Theorem.** Let P be a predicate and $M_P$ its corresponding finite state automata. P is consistent iff $L(M_P) \neq \varnothing$.

**Proof.** Assume P is consistent. By definition there exists a nonempty relation R s.t. R satisfies P. It follows from the previous definition that if R satisfies P then $R \subseteq L(M_p)$ and hence $L(M_P)$ is not empty.

Similarly, if $L(M_P) \neq \varnothing$, consider some element $a \in L(M_p)$. Since there is a one-to-one correspondence between tuples in $A_1 \times A_2 \ldots \times A_n$ and accepting strings (and hence elements of $L(M_P)$), define R to be a relation consisting of the tuple t corresponding to a. Then R satisfies P and hence P is consistent.

□

This theorem allows for predicate consistency checking because of the following result.

**Theorem.** Let M be a f.s.a. Then the predicate $L(M)=\varnothing$ is decidable.

**Proof.** See Sudkamp [14] for an algorithmically constructive proof. □

The above framework may be utilized to check for equivalence of alternate constraint sets. This is the result of the next theorem.

**Theorem.** Let $P_1$ and $P_2$ be sets of predicates and $M_{P_1}$ and $M_{P_2}$ be their corresponding f.s.a. $P_1$ is equivalent to $P_2$ iff $L(M_{P_1})=L(M_{P_2})$.

**Proof.** Assume $P_1$ and $P_2$ are equivalent. First show that $L(M_{P_1})\subseteq L()$. Let a $\in L(M_{P_1})$ and R the relation consisting of its corresponding tuple. Then, R satisfies $P_1$ and hence R satisfies $P_2$. Therefore, a, the corresponding string, is a member of $L(M_{P_2})$. Similarly, we can show that $L(M_{P_2})\subseteq L(M_{P_1})$, and therefore, are equal.

Now assume $L(M_{P_1})=L(M_{P_2})$. Let R be a relation corresponding to some subset S of $L(M_{P_1})$. Then R satisfies $P_1$. But, S is a subset of $L(M_{P_2})$. Hence, R satisfies $P_2$. Therefore, $P_1$ and $P_2$ are equivalent. □

Since predicate equivalence implies language equivalence we may now implement an algorithm to test for predicate equivalence based on the following theorem.

**Theorem.** Let $M_1$ and $M_2$ be two f.s.a and $L(M_1)$ and $L(M_2)$ be their corresponding languages. Then, $L(M_1) = L(M_2)$ iff

$$L = (L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)}) \cap L(M_2))$$

is empty.

**Proof.** If $L(M_1) = L(M_2)$, then the intersection of a language and its complement is empty, and thus, L is empty.

Now assume that

$$L = (L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)}) \cap L(M_2))$$

is empty.

The proof that $L(M_1) = L(M_2)$ is by contradiction. If $\in L(M_1)-L(M_2)$, then $a\in L(M_1))\cap \overline{L(M_2)})$ Therefore, the antisymmetric union is nonempty. Similarly, for $a\in L(M_2))\cap L(M_1)$. □

The above antisymmetric difference can be associated with a f.s.a. as discussed above. Hence, by the algorithm to test for emptiness of a f.s.a we may test for equivalence of constraint sets.

Finally, the technique above allows for a straightforward determination of redundancy in a constraint set. Let $\boldsymbol{P} = \{P_1 \ldots P_n\}$ be a set of constraint predicates. The following algorithm will test this given set $\boldsymbol{P}$ for redundancy and produce a nonredundant subset of P.

{Begin Algorithm REDUN}

1. Redundant := *false*; $P_{CURR} := \boldsymbol{P}$;
2. For $P_i \in P$ do

Begin

Let $Q := P_{CURR} - \{P_i\}$.

If $P_{CURR}$ is equivalent to Q, then $P_{CURR} := Q$.

End;

3. If $P_{CURR} \neq P$ then Redundant := *true*.

{End of Algorithm REDUN}

Notice that in step 2 the equivalency of $P_{CURR}$ and Q is computed using the results of the two previous theorems. The correctness of the above algorithm is proven in the following theorem.

**Theorem.** At the termination of the algorithm REDUN the variable $P_{CURR}$ contains a nonredundant subset of $\boldsymbol{P}$ which is equivalent to $\boldsymbol{P}$.

**Proof.** Denote by $P_{EXIT}$ the set $P_{CURR}$ at the termination of the algorithm REDUN. It is clear that $P_{EXIT}$ is equivalent to $\boldsymbol{P}$ because every time $P_{CURR}$ is reset to a new value Q in step 2, this Q is equivalent to $P_{CURR}$.

To show that $P_{EXIT}$ is nonredundant assume the opposite is true. Then, there exists a proper subset $P'_{EXIT} \subset P_{EXIT}$ such that $P'_{EXIT}$ is equivalent to $P_{EXIT}$. Let $P_i \in P_{EXIT}-P'_{EXIT}$. Notice that $P_{EXIT} \subseteq P_{CURR}$ and $P_{EXIT} \subseteq Q$ for all iterations of step 2. Hence, $P_i \in P_{CURR}$ for all iterations. Since $P_i \in P$, at some iteration of step 2, we have $Q := P_{CURR}-\{P_i\}$. For this Q we also have $P'_{EXIT} \subseteq Q \subseteq P$. But, $P'_{EXIT}$ is equivalent to $P_{CURR}$ and hence equivalent to $\boldsymbol{P}$ and Q. Therefore, at this iteration $P_{CURR}$ is reset to Q, which does not contain $P_i$. But, this contradicts the fact that $P_{EXIT}$ contains $P_i$ since nowhere in the algorithm do we add elements to $P_{CURR}$.

## 6. Conclusion

This paper describes a way to translate semantic requirements for database integrity into the syntactic language of regular expressions. This, in turn, provides a way to associate a finite state automata to the sets of predicates defining the integrity constraints. These machines allow for the real time processing of database update records in many important application areas. A tuple of size n can now be processed in O(n) time. In addition, this approach leads to efficient techniques for dealing with the problems of consistency, equivalence and redundancy of integrity constraints.

## References


1. **A. V. Aho, R. Sethi, and J.D. Ullman,** *Compilers: Principles, Techniques and Tools,* Addison-Wesley:Reading, Mass, 1988.

2. **S. Baase,** *Computer Algorithms: Design and Analysis,* Addison-Wesley:Reading, Mass, 1988.

3. **E. F. Codd,** "Relational Completeness of Data Base Sublanguages," *Courant Computer Science Symposia b,* in **Data Base Systems,** Prentice Hall, 1971.

4. **E. F. Codd,** *The Relational Model for Database Management : Version 2,* Addison Wesley, Pennsylvania, 1990.

5. **P. J. Denning, J.B. Dennis and J.E. Qualitz,** *Machines, Languages, and Computation,* Prentice Hall, New Jersey, 1978.

6. **R. Elmasri and S.B. Navathe,** *Fundamentals of Database Systems,* Benjamin/Cumming Publishing Company, Inc., 1989.

7. **K. P. Eswaran** *et al,* "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM,* Volume 19, Number 11, November, 1976.

8. **G. Gardarin and P. Valduriez,** *Relational Databases and Knowledge Bases,* Addison Wesley, Pennsylvania, 1989.

9. **M. M. Hammer,** "A Framework for Data Base Semantic Integrity," MIT Laboratory for Computer Science, Cambridge, Mass., 1976.

10. **J. E Hopcroft and J.D. Ullman,** *Formal Languages and their Relation to Automata,* Addison-Wesley, Pennsylvania, 1979.

11. **J. G. Hughes,** *Database Technology : A Software Engineering Approach,* Prentice Hall International Series in Computer Science, New Jersey, 1988.

12. **K. Parsaye, M. Chignell, S. Khoshafian, and H. Wong,** *Intelligent Databases,* J. Wiley & Sons Inc., 1989.

13. **Y. C. Sagiv,** *Optimization of Queries in Relational Databases,* UMI Research Press, 1981.

14. **T. Sudkamp,** *Languages and Machines,* Addison-Wesley, 1989.

15. **J. D. Ullman,** *Principles of Database and Knowledge-based Systems,* Vol. 1, Computer Science Press, Maryland, 1988.



**Robert R. Goldberg** is Assistant Professor at Queens College. He received his Ph.D. in Computer Science from New York University. His research interests are in Computer Theory and Computer Vision.

**Jacob Shapiro** is Associate Professor at Baruch College. He received his Ph.D. in Mathematics from UC-Berkeley. His research interests are Databases, Networks and Algebraic Systems.

**Jerry Waxman** is Associate Professor at Queens College. He received his Ph.D. in Computer Science from New York University. His research interests are Networks, Computer Theory, Voice Recognition and Computer Education.



*Received:* December, 1992
*Accepted:* August, 1993



*Contact address:*
Robert R. Goldberg
Department of Computer Science
Queens College of CUNY
65 - 30 Kissena Blvd.
Flushing, New York 11367-0904
Email: goldberg@qcunix.acc.qc.edu