A Step Towards the Automatic Maintenance of the Semantic Integrity of Databases

R. A. Frost and S. Whittaker
Department of Computer Science, University of Strathclyde, Livingstone Tower, 26 Richmond Street, Glasgow G1 1XH, UK

A database should be an accurate model of that part of the universe which it represents. However, this is rarely achieved. Errors in the data occur for a variety of reasons. Various techniques have been developed to reduce such errors and/or to detect them when they occur. Some of these techniques are concerned with checking that the data is 'sensible', i.e. that is complies with certain constraints which are derived from our knowledge of the semantics of that part of the universe which is represented by the data. Such constraints are called semantic integrity constraints; an example is: 'no-one may be their own father'. Ideally, a database implementor should be able to specify a set of semantic constraints and then let the system enforce them automatically. Some progress has been made towards this ideal. However, the constraint definition languages which have been developed tend to be syntactically complex, and the enforcement of constraints is often carried out in an *ad hoc* manner. An alternative approach is proposed in this paper. We present a simple constraint definition language, SCHEMAL, and show how constraints expressed in it can be enforced automatically. We describe in detail the algorithm which is used to enforce SCHEMAL constraints. The method has been fully implemented at the University of Strathclyde.

1. INTRODUCTION

1.1 The problem

Ideally, a database should be an accurate model of that part of the universe which it represents. However, this situation is rarely achieved. Errors in the data occur for various reasons:

(i) Parts of the universe are observed incorrectly before they are represented by data. For example: a thermometer might not be calibrated correctly.
(ii) Observations are not made frequently enough. For example: someone might be represented as having single status in the database, whereas in reality they have just married.
(iii) Data are corrupted in coding, transcription, and/or transmission.
(iv) Data are corrupted in storage through hardware faults.
(v) Errors are introduced through concurrent update. For example: user 1 reads balance B at time $T1$, adds X to B and overwrites the original balance at time $T2$. User 2 repeats the process, reading at time $T3$, adding Y to the balance, and writing at time $T4$. If $T1 < T3 < T2 < T4$ the balance will be in error if $X \neq 0$.

Various techniques have been developed to reduce the occurrence of such errors, or to detect them when they occur. Many of these techniques are well understood and their use is common practice. For example: (i) parity checks are used to detect errors in data transmission, and (ii) 'locks' are used to reduce errors due to concurrent update. A concise description of some of these methods is given by Wiederhold.[1]

Some aspects of the problem are very poorly understood. For example: the problem of 'out-of-date' data has not received much attention in the literature. However, Klopprogge[2] has addressed this question in a recent paper and makes some interesting suggestions as to how the time dimension might be taken into account.

Other aspects of the problem are still poorly understood although some progress has been made over the last few years. One of these aspects is concerned with the *semantic integrity* of the database. By semantic integrity we mean the compliance of the database with constraints which are derived from our knowledge about what is and is not 'allowed' in that part of the universe which is represented by data.

The *maintenance* of semantic integrity involves preventing data which represent a disallowed state of the universe from being inserted into the database. (These rather woolly descriptions of semantic integrity will suffice for the present. We give more formal definitions later.) The routines which maintain semantic integrity detect 'semantic errors' (disallowed states), inform the user that an error has taken place, and prevent data from being inserted into the database.

This paper is concerned with the *automatic* maintenance of database integrity, and in particular is concerned with the detection of semantic errors. We commence by reviewing past work.

1.2 Past work

A straightforward approach for the maintenance of semantic integrity follows from suggestions by Abrial[3] and Florentin.[4] It involves two steps:

(i) Provide the database implementor with a language for specifying semantic constraints.
(ii) Use the constraints to vet data which are presented for insertion into the database.

Ideally, the database implementor should be able to specify constraints which are then used *by the system* to maintain semantic integrity automatically. However, this approach must be viewed with caution: First, for a general language it is undecidable whether the constraints

CCC-0010-4620/83/0026-0124 $04.50

expressed in it are themselves consistent. Second, vetting data against a complex integrity constraint might require access to a large portion of the database.

The first problem can be solved only if the language in which the constraints are expressed is simple (e.g. it must not contain function symbols) and well defined. The second problem is related to the first. Given a constraint, the system must be capable of determining how costly it is to apply that constraint. Ideally, this cost should be estimated when the constraints are being formulated, thereby allowing the user to replace 'expensive' constraints with 'cheaper' ones. We illustrate this with an example taken from Blaser and Schmutz:[5] given a 'father' relation, a semantic constraint might state that the subgraph containing only edges labeled with 'father' must be cycle free. Checking this constraint is likely to be expensive. If the database also contains birthdates, then the constraint above could be replaced by a constraint which states that the birthdate of the father must precede the birthdate of the child.

These considerations, however, have not deterred the development of the two step approach, and progress has been achieved through the work of Abrial,[3] Florentin,[4] Eswaran and Chamberlin,[6] Stonemaker,[7] Hammer and McLeod,[8] Biller and Neuhold,[9] Pelagatti, Paolini and Bracchi,[10] Roussopoulos,[11] Mylopoulos, Bernstein and Wong,[12] Brodie,[13] Borgida and Wong[14] and Shipman.[15]

Florentin[4] was one of the first to propose that predicate calculus could be used for the precise specification of semantic integrity constraints. However, he goes on to state that the manipulation of logical formulae is likely to be too tedious to be useful as a practical method for the maintenance of semantic integrity. This conclusion is accompanied by a reference to work carried out by Robinson.[16]

The papers by Eswaran and Chamberlin[6] and Stonebraker[7] are concerned with the maintenance of semantic integrity of relational databases. Constraints are specified in two relational query languages: SEQUEL and QUEL, respectively. However, no indication of how such contraints might be enforced is given.

A more general approach is claimed to have been made by Hammer and McLeod.[8] Although they describe their work in the context of a relational database system, they go some way towards the specification of a special purpose constraint definition language. Several useful examples of constraints are presented in Hammer and McLeod's paper, but no indication of how these constraints might be enforced is given.

Biller and Neuhold[9] also recognize the need for generality and point out that it would be difficult to translate an integrity constraint formulated in terms of one specific view (e.g. the relational view) into constraints expressed in terms of some other view (e.g. the hierarchical view). Consequently, they have developed an abstract conceptual view which uses the concepts of entity, entity type, and relation. Constraints are expressed in terms of this view in a language called LDDL. Although LDDL has a simple semantic basis, it has a complex syntax. For example, there are seven different kinds of 'type' definition. No indication of how LDDL constraints might be enforced is given.

Pelagatti, Paolini and Bracchi[10] discuss semantic integrity in relation to their main topic of 'mapping external views to a common data model'. For this reason they restrict their attention to two types of constraint: 'cardinality' and 'equivalence' constraints. They suggest a technique similar to that proposed by Abrial[3] for expressing cardinality constraints and claim that the consistency of cardinality constraints can be checked automatically, but no indication of how they might be enforced is given.

The next significant contribution was made by Roussopoulos[11] who developed a conceptual schema definition language called CSDL. This language is based on the semantic-network model of databases proposed by Roussopoulos and Mylopoulos[17] and as such is more 'data-independent' than earlier languages (i.e. it is independent of the structures which are used to store the data).

CSDL facilitates the specification, modification, and examination of constraints through high level support facilities. However, once again, no indication of how constraints might be enforced is given.

Another major step forward was made by Mylopoulos, Bernstein and Wong[12] in their design of a language called TAXIS. This language is specifically aimed at interactive information systems such as airline booking systems. It offers traditional database management facilities and a means of specifying semantic integrity constraints, integrated into a single language. The conceptual framework underlying TAXIS is based on the concepts of class, property, and the 'IS-A' hierarchy similar to that proposed by Smith and Smith.[18] Semantic integrity constraints are regarded as prerequisite and result conditions of transactions which must be met if the effect of the transaction is to be accepted. Mylopoulos *et al.* give a few examples of semantic integrity constraints but do not indicate how they might be enforced. Their main contribution is in recognizing that constraints defined over one class of entities or transactions should be automatically inherited by sub-sets of that class.

Borgida and Wong[14] have given two formal specifications of the semantics of the TAXIS language one of which is based on axioms and partial correctness assertions intended for verifiers who wish to show that the system maintains database integrity. However, they do not describe the mechanics by which the system actually could maintain integrity.

A somewhat different approach has been proposed by Brodie[13] who suggests that data type tools can be used for the definition of semantic constraints and for the maintenance of semantic integrity. Strong typing is provided in a Pascal-like type system embedded in a language called BETA. For example, specification of the type 'person object' might be:

```
type
  person = object name         : nametype;
                  number       : social-insurance #;
                  sex          : (male, female)
                                 unordered;
                  address      : address type;
                  title        : (professor, tutor,
                                 student);
                  keys           name, number
                  dependencies names→
                                 address|sex|title
           end object;
```

Enforcement of the semantic constraints implied by this

specification may then be carried out automatically by normal type-checking routines. For example the constraint: 'person objects may only be of sex male or female', which is implied in the above, is easily enforced.

However, it is not easy to see how more complex constraints such as 'a person may not be the son of someone born at a later date than themselves' could be expressed in a type declaration. Brodie's work is interesting because it questions the values which are used to distinguish between syntax and semantics.

Shipman[15] has designed a data definition and manipulation language called DAPLEX. This language is based on a conceptual view called the 'functional' view which is similar to the binary-relational view in some respects. Constraints may be expressed in DAPLEX as, for example:

```
DEFINE CONSTRAINT Native Head (Department) ⇒
                  Dept (Head(Department)) = Department
```

which states that a department's head must come from within the department.

A complete implementation of DAPLEX is not yet available. However, an implementation written in ADA, called ADAPLEX, should be available shortly.[19]

1.3 Review of past work

Most of the approaches described in Section 1.2 have concentrated on the *specification* of integrity constraints rather than on the *enforcement* of these constraints. Many of the methods classify constraints in what would appear to be an attempt at facilitating specification. This has lead to syntactically complex languages and presumably *ad hoc* methods for the enforcement of constraints.

In no case has a detailed description of how constraints might be enforced been presented. We must assume either (i) no method exists or (ii) the method is too complex to be outlined in a publishable paper.

1.4 An alternative approach

The approach which we have adopted and which is described in this paper concentrates on constraint *enforcement* rather than on constraint specification. We have designed a constraint definition language called SCHEMAL such that *any constraint which can be expressed in SCHEMAL can be enforced automatically by a general purpose algorithm.* Constraint enforcement is relatively straightforward and we describe the algorithm in detail.

We do not claim that our approach represents a complete solution to the problem. SCHEMAL is based on a decidable sub-set of first order predicate calculus and therefore inherits all of the expressive limitations of this logical framework. The main purpose of this paper is to demonstrate that the use of a simple conceptual framework as a basis for the specification of constraints can lead to a simple method for their enforcement. We see SCHEMAL as the first in a series of languages. The next step will be to apply a similar approach and to develop a more powerful language based on a sub-set of a higher order logic. We discuss this further in the last section of the paper.

The sub-set of first order predicate calculus which underlies SCHEMAL is called the 'simplified binary-relational (SBR) view of the universe'.[20] It does not contain functions and only accommodates binary relations. A database based on the SBR view is called an SBR database. SCHEMAL and the method for maintaining database integrity are described in relation to an SBR database for which it was developed. However, the approach could be readily adapted for other types of database.

The SBR view is described in Section 3, SCHEMAL is described in Section 4, and the method for the automatic maintenance of database integrity is given in Section 5.

2. EXAMPLES OF INTEGRITY CONSTRAINTS

We shall use the following constraints as examples in this paper:

C1: members of the set of husbands may not also be members of the set of bachelors.
C2: only students may enrol for something.
C3: an employee's manager must also be an employee.
C4: professors may not earn more than £20 000.
C5: male persons are only allowed to marry if they are older than 18 years.
C6: no two entities may have the same name and address.
C7: a department can only have one manager.
C8: no one can set an exam unless they teach someone who is a student.
C9: all employees who work in the same department must have the same personal manager.
C10: the only relationship in which lecturers are allowed to participate are of type 'teaches' and 'sets exam'.
C11: a married person may not become a single person.
C12: an employee's salary may not be reduced.
C13: the professor's average salary may not exceed £16 000.
C14: an employee's salary must exceed deductions.

Constraint C2 is interpreted as meaning that an entity may not be enrolled for a course unless that entity is *known* to be a member of the set of students. For example, the data 'entity #3 is enrolled for English' may not be inserted into the database unless that database already contains the data 'entity #3 is a member of the set of students'. Similar interpretations apply to constraints C3, and C5.

C11 and C12 constrain the way in which the database may be updated rather than extended.

3. THE SIMPLIFIED BINARY-RELATIONAL VIEW OF THE UNIVERSE

The simplified binary-relational (SBR) view of the universe regards the universe as consisting of *entities* with *binary relationships* between them.[20] An entity is anything of interest which can be identified. A binary relationship is an association linking two entities or an entity to itself.

A *binary relation* is a set of binary relationships. *N*-ary relationships, such as '*a* bought *b* from *c*' are reduced to sets of binary-relationships by the explicit naming of the implied entity involved. For example:

sale #1. buyer is. *a*
sale #1. item bought. *b*
sale #1. seller is. *c*

$\in$, the set-membership relation, is treated like any other binary relation. For example, the fact, that 'John is a policeman', is regarded as a binary relationship of type $\in$ linking the entity 'John' to the entiry 'set of policemen':

John. $\in$. policemen

Things and properties of things are both regarded as entities. For example, the fact that 'John is 6′ 1″ tall' is regarded as:

John. height is. 6′ 1″

In order to rationalize the semantic rules which might apply in some part of the universe, the SBR view includes additional concepts denoted by:

variable, (–. –. –), $\Leftarrow$, $\wedge$, $\vee$, $\neg$, [. .]

where:

(i) a *variable* is a universally quantified variable which is used to represent entities. In the remainder of this paper, variables are represented by single capital letters.
(ii) (*X.Y.Z*) means that a relationship of type *Y* exists between entities *X* and *Z*. This is similar to the logical construct *Y*(*X*, *Z*) in which *Y* is a predicate. In agreement with Kowalski,[21] the infix notation is regarded as more natural.
(iii) $P \Leftarrow Q$ means *P* if *Q*
(iv) $\wedge$ is the logical conjunction operator: AND
(v) $\vee$ is the logical disjunction operator: OR
(vi) $\neg$ is the logical negation operator: NOT
(vii) [. .] are brackets with usual meaning

These concepts are sufficient to describe many, *but not all*, semantic rules. For example, the rule 'someone may not be a bachelor and a husband' may be expressed as

$$\neg(X. \in. \text{bachelors}) \Leftarrow (X. \in. \text{husbands})$$

A set of data representing a binary relationship is called a triple and consists of three fields. A set of triples is called an *SBR database* and may be stored in a structure called an SBR data storage structure. Triples may be retrieved from such structures using any combination of fields as key. Retrieval requests are denoted as, for example:

retrieve (?. $\in$. people)

which retrieves all members of the set of people.

SBR storage structures are similar to binary-relational storage structures as described by Frost.[22]

4. SCHEMAL: A CONCEPTUAL SCHEMA CONSTRAINT LANGUAGE

A *conceptual schema* is a set of semantic rules which apply to some part of the universe. Conceptual schemas do not refer to data nor to its manipulation. However, the rules may be used:

(i) to constrain what may be represented by data
(ii) to infer unknown aspects of the universe from known aspects and, therefore, to infer new data from existing data.

SCHEMAL[20] is a conceptual schema constraint language which is based on the SBR view. SCHEMAL is a decidable sub-set of first-order predicate calculus. By decidable, we mean that the consistency of a set of SCHEMAL rules can be determined. An implementation of SCHEMAL has been carried out by Asher.[23] The formal syntax of SCHEMAL is given in Figure 1.

⟨rule⟩ : : = ⟨consequent⟩ ⇐ ⟨antecedent⟩ | ⟨consequent⟩ ⇐ | ⇐ ⟨antecedent⟩
⟨consequent⟩ : : = ⟨antecedent⟩
⟨antecedent⟩ : : = ⟨conjunction⟩ | ⟨disjunction⟩ | ⟨expression⟩
⟨conjunction⟩ : : = ⟨expression⟩ { ∧ ⟨expression⟩ }
⟨disjunction⟩ : : = ⟨expression⟩ { ∨ ⟨expression⟩ }
⟨expression⟩ : : = ¬⟨item⟩ | ⟨item⟩
⟨item⟩ : : = ⟨triple⟩ | [⟨conjunction⟩] | [⟨disjunction⟩]
⟨triple⟩ : : = (⟨entvar⟩ . ⟨relation⟩ . ⟨entvar⟩)
⟨entvar⟩ : : = ⟨entity⟩ | ⟨variable⟩

Figure 1. The formal syntax of schemal.

Rules C1, C2 and C6 of Section 2 may be expressed in SCHEMAL as:

C1′: $\neg(X. \in. \text{bachelors}) \Leftarrow (X. \in. \text{husbands})$
C2′: $(X. \in. \text{students}) \Leftarrow (X. \text{enrolled for. } Y)$
C6′: $(X. \text{ident. } Y) \Leftarrow [(X. \text{named. } N) \wedge (X. \text{lives in. } A) \wedge (Y. \text{named. } N) \wedge (Y. \text{lives in. } A)]$

These may be read as: 'if an entity is a member of the set of husbands, then it must not be a member of the set of bachelors', 'if an entity is enrolled for anything, then it must be a member of the set of students', and 'if two entities have the same name and address then they must be identical'.

SCHEMAL is limited in its expressive power. For example, it cannot be used to specify constraints such as C10, C13, or C14. Also, it is not yet clear how it might be used in relation to constraints such as C11 and C12. However, most rules which can be specified in SCHEMAL can be used to maintain database integrity automatically. The next section illustrates how this is done.

5. AUTOMATIC MAINTENANCE OF DATABASE INTEGRITY

5.1 Informal overview of the method

An SBR database exhibits semantic integrity with respect to a set of SCHEMAL constraints if all (possibly partial) *instantiations* of these constraints which are *implied by* the triples in the database are true. If any implied instantiation is false then the database does not exhibit semantic integrity.

An implied instantiation of a constraint is derived by the substitution of variables in that constraint by entities.

For example, the triple (e_1. ∈. bachelors) implies the following instantiation of constraint C1:

$$\neg(e_1.\in.\text{bachelors}) \Leftarrow (e_1.\in.\text{husbands})$$

A formal definition of an *implied instantiation* is given in subsection 5.4.

The integrity of a database can be *maintained* by checking that all *new* instantiations of constraints which are implied by a new triple would be true if the triple were in the database. If so, the triple may be inserted into the database, otherwise it is rejected.

Maintenance of database integrity involves a five-stage process:

(i) the constraints are specified in SCHEMAL by the database implementor. For example:

$$C2': (X.\in.\text{students}) \Leftarrow (X.\text{enrolled for.}\ Y)$$

(ii) The SCHEMAL constraints are then converted to a partially simple form called *clausal form*, which is described in section 5.2. An example of a rule in clausal form is:

$$C2'': (X.\in.\text{students}) \vee \neg(X.\text{enrolled for.}\ Y)$$

which may be read as 'either X is a student or X is not enrolled for anything'.

(iii) On attempting to insert a new triple TI into the database, it is *matched* against the constraints in clausal form, as discussed in section 5.3. For example, the triple (e_1. enrolled for. physics) matches constraint C2″ above.

(iv) For every constraint C which is matched by TI, the set of *new* (possible partial) instantiations of C which are implied by TI and the database are generated. For example, the set of *new* instantiations of C2″ above, implied by the triple (e_1. enrolled for. physics), consists of one instantiation only, irrespective of the database contents:

$$(e_1.\in.\text{students}) \vee \neg(e_1.\text{enrolled for. physics})$$

(v) The new instantiations are then evaluated *assuming that the new triple were in the database*. If the instantiations are *all* true then the triple may be put into the database, otherwise it is rejected. For example, the triple (e_1. enrolled for. physics) may only be inserted if the triple (e_1. ∈. students) is already in the database. If this triple is not in the database then the instantiation of C2″ above is false.

We now describe this process more precisely, define the terms introduced, and show how stages (ii) to (v) may be carried out automatically.

5.2 Conversion of constraints to clausal form

A SCHEMAL constraint in *clausal form* consists of a disjunction of *literals*, where a literal is a triple or the negation of a triple. Examples of constraints in clausal form are:

$$C1'': \neg(X.\in.\text{bachelors}) \vee \neg(X.\in.\text{husbands})$$

which may be read as 'either X is not a bachelor or X is not a husband'

$$C2'': (X.\in.\text{students}) \vee \neg(X.\text{enrolled for.}\ Y)$$

$$C6'': (X.\text{ident.}\ Y) \vee \neg(X.\text{named.}\ N) \vee \neg(X.\text{lives in.}\ A) \vee \neg(Y.\text{named.}\ N) \vee \neg(Y.\text{lives in.}\ A)$$

Some SCHEMAL constraints transform into more than one constraint in clausal form. For example:

$$[\neg(X.\in.\text{animals}) \wedge \neg(X.\in.\text{minerals})] \Leftarrow (X.\in.\text{vegetables})]$$

transforms into the two constraints:

$$\neg(X.\in.\text{animals}) \vee \neg(X.\in.\text{vegetables})$$

$$\neg(X.\in.\text{minerals}) \vee \neg(X.\in.\text{vegetables})$$

Automatic conversion of SCHEMAL constraints is described by Whittaker[24] and is based on a method given by Nilsson.[25]

5.3 Matching a triple against a constraint

The method which we are describing assumes that all triples in the database are variable-free. This assumption is reasonable for most database applications.

A variable-free triple TI *matches* a constraint C if there is at least one triple TC in C which corresponds to TI. Two triples TC and TI correspond if:

(i) TI = TC

or

(ii) TI and TC have identical second fields

and

- (a) TI and TC have identical first fields AND the third field of TC is a variable

or

- (b) TI and TC have identical third fields and the first field of TC is a variable

or

- (c) the first and third fields of TC are variables

For example:

(e_2. ∈. bachelors) matches C1″ above
(e_3. named. Peter) matches C6″ above

5.4 Implied instantiations

An *instantiation* of a constraint is obtained by the consistent substitution of *all* of its variables by entities.

A *partial instantiation* of a constraint is obtained by the consistent substitution of *some* of its variables by entities. For example, the following is a partial instantiation of constraint C2″:

$$(e_4.\in.\text{students}) \vee \neg(e_4.\text{enrolled for.}\ Y)$$

The set of new (possibly partial) instantiations of a constraint C, which is implied by matching triple TI, is generated by calling the recursive procedure *p-instant* below for each triple TC in C which corresponds to TI. Notice that the set generated will depend upon the database into which TI is being proposed for insertion.

Notes

(i) **constraint** C is an array of literals representing a constraint. For example:

(e_1. ∈. students)	¬(e_1. enrolled for. Y)

(ii) TDX is also an array of literals, none of which contains a negation sign.
(iii) The statement commencing 'substitute variables in TC by equivalent . . .' is not a comment on the code following it, but is shorthand for a number of statements, details of which are not given.
(iv) The procedure generates all *new* instantiations of C which are implied by TI due to its correspondence to TC in C.
(v) The Boolean variable *sub* is used to flag when no more variable substitution can be made. At this point a new (possibly partial) instantiation has been generated and is printed. The call *print* is replaced later by a procedure call to evaluate the instantiation.
(vi) [] **triple** TDX := *retrieve* (TX); causes the set of triples which correspond to TX to be retrieved from the database and assigned to TDX.

```
proc p-instant (triple TI, TC, constraint C):
begin
substitute variables in TC by equivalent entities in TI;
make the same substitution(s) throughout C;
bool sub := false;

for each triple TX in C still containing one or two
variables
do begin
  [] triple TDX := retrieve (TX);
  if TDX isnt empty
    then sub := true;
         for each TD in TDX
           do begin
             p-instant (TD, TX, C)
              end
  fi

  end;
if sub = false
  then print (C)
fi

end;
```

Dr P. M. D. Gray of Aberdeen University has pointed out that the algorithm above is actually a variant of the recursive formulation of the eight queens problem.[26] The vector of constraints corresponds to the chessboard and instantiation of variables to placing of queens in positions.

Dr Gray also noted that the algorithm is very similar to the process of unification in theorem proving and in particular is similar to the method used to do this in the PROLOG language.[27] If the database were held as PROLOG unit clauses, then the constraint could be held directly in PROLOG form, for example:

```
C2 (x):– not ((not(tup(exist, x, studs)),
                                  tup(enroll, x, y)))
```

PROLOG is more powerful than SCHEMAL in that it can handle function symbols and will evaluate arithmetic expressions. However, it should be noted that the consistency of a set of constraints which contain function symbols is undecidable as pointed out by Frost *et al.*[20]

The initial call of *p-instant* substitutes the entities from the triple to be inserted into corresponding variables in C. Subsequent recursive calls substitute the remaining variables in all possible ways which are implied by triples in the database. Sometimes, it may not be possible to substitute all the variables. In these cases, partial instantiations are generated. To illustrate the process, consider the following examples:

Suppose that the database contains the triples:

$$\begin{array}{l}(e_1.\quad \in \quad .\text{students})\\ (e_1.\ \text{named}\quad .\text{Peter}\quad)\\ (e_1.\ \text{lives in}\quad .\text{London})\\ (e_2.\ \text{named}\quad .\text{Peter}\quad)\\ (e_3.\ \text{lives in}\quad .\text{London})\\ (e_4.\ \text{named}\quad .\text{Peter}\quad)\end{array}$$

and also suppose that the following constraints hold:

$$C2'': (X.\in.\text{students}) \vee \neg(X.\text{enrolled for}.\ Y)$$
$$C6'': (X.\text{ident}.\ Y) \vee \neg(X.\text{named}.\ N) \vee \neg(X.\text{lives in}.\ A) \vee \neg(Y.\text{named}.\ N) \vee \neg(Y.\text{lives in}.\ A)$$

Example 1. If TI = (e_4. ∈. students) then this matches C2″ because TI corresponds to (X. ∈. students). With TC = (X. ∈. students), the call *p-instant* (TI, TC, C2″) generates the single partial instantiation:

$$\text{I1}: (e_4.\in.\text{students}) \vee \neg(e_4.\text{enrolled for}.\ Y)$$

No other triples in C2″ correspond to TI, therefore no further substitutions can be made, consequently I1 is the only new instantiation of C2″ which is implied.

Example 2. If TI = (e_2. enrolled for. physics) then this matches C2″ because TI corresponds to TC = (X. enrolled for. Y). The call *p-instant* (TI, TC, C2″) generates the instantiation:

$$\text{I2}\ (e_2.\in.\text{students}) \vee \neg(e_2.\text{enrolled for. physics})$$

No other triples in C2″ correspond to TI, therefore, this is the only instantiation implied.

Example 3. If TI = (e_5. lives in. London) then this triple matches C6″ because TI corresponds to TC = (X. lives in. A). The call *p-instant* (TI, TC, C6″) generates the instantiations:

$$\begin{array}{ll}\text{I3}: (e_5.\text{ident}.\ e_5) & \vee \neg(e_5.\text{named. Peter}) \vee \neg(e_5.\text{lives in. London})\\ & \vee \neg(e_5.\text{named. Peter}) \vee \neg(e_5.\text{lives in. London})\\ \text{I4}: (e_5.\text{ident}.\ e_2) & \vee \neg(e_5.\text{named. Peter}) \vee \neg(e_5.\text{lives in. London})\\ & \vee \neg(e_2.\text{named. Peter}) \vee \neg(e_2.\text{lives in. London})\\ \text{I5}: (e_5.\text{ident}.\ e_4) & \vee \neg(e_5.\text{named. Peter}) \vee \neg(e_5.\text{lives in. London})\\ & \vee \neg(e_4.\text{named. Peter}) \vee \neg(e_4.\text{lives in. London})\\ \text{I6}: (e_5.\text{ident}.\ e_3) & \vee \neg(e_5.\text{named. Peter}) \vee \neg(e_5.\text{lives in. London})\\ & \vee \neg(e_3.\text{named. Peter}) \vee \neg(e_3.\text{lives in. London})\end{array}$$

The triple (e_5. lives in. London) also corresponds to triple (Y. lives in. A) in C6″. The set of instantiations implied by this correspondence is similar to the set above.

Notice that the set of new instantiations of C is implied by a triple *in conjunction with the database* for which the triple is being tested.

5.5 Evaluation of instantiations

For reasons which will be apparent later, we consider the evaluation of two types of instantiation:

5.5.1 Evaluation of instantiations in which no further variable substitution can be made. These instantiations may be completely variable-free or may contain triples with variables for which there is no corresponding triple in the database, e.g. I1 of Section 5.4.

Such (possibly partial) instantiations are true if they contain at least one literal which is true.

A literal which contains a negation sign is true if the triple it contains is false otherwise it is false. A literal which does not contain a negation sign is true if the literal it contains is true otherwise it is false. For example, if the triple (e_1. ∈. students) is true then the literal (e_1. ∈. students) is true and the literal ¬(e_1. ∈. students) is false.

A triple is assumed to be true: (i) if it is the triple being tested for input, or (ii) if a *corresponding* triple is stored in the database, or (iii) if its evaluation is true. Triples such as (John. ident. John) and (5. <. 3) can be evaluated to produce true or false.

The assumption here is that if a triple is missing from the database or is not being proposed for input then the corresponding fact is false. This is the 'closed world assumption' as discussed by Minker.[28]

From the above, it can be seen that instantiation I1 is true, and I2 is false with respect to the example database of Section 5.4.

5.5.2 Evaluation of instantiations where further variable substitution is possible. Such instantiations are generated as intermediate results by *p-instant* and are not printed out. For example, in the derivation of I5, the following instantiation is generated as an intermediate result:

$$(e_5.\ \text{ident.}\ Y) \vee \neg(e_5.\ \text{named.}\ N) \vee \neg(e_5.\ \text{lives in.}\ \text{London}) \vee \neg(Y.\ \text{named.}\ N) \vee \neg(Y.\ \text{lives in.}\ \text{London})$$

An *intermediate* partial instantiation is true *irrespective of further variable substitution* if either:

(i) It contains at least one literal with no negation sign and a *variable-free* triple which is true.

or

(ii) It contains at least one literal with a negation sign and a triple (which may or may not be variable-free) which is false.

Similarly, an intermediate partial instantiation is false irrespective of further variable substitution if it contains no literals which could become true by such substitution. That is if *each one* of its literals is either:

(i) A literal with no negation sign and with a triple (which may or may not be variable-free) which is false.

or

(ii) A literal with a negation sign and with a *variable-free* triple which is true.

5.6 Accepting or rejecting triples

As mentioned earlier, the integrity of a database D, with respect to a set of constraints C, can be maintained by checking each triple TI, proposed for input, to make sure that all *new* instantiations of C, which are implied by TI and D would be true if TI were in the database. So far, we have described methods which could be used to carry out this checking automatically. However, such an approach assumes that the database already exhibits semantic integrity which is then maintained. A simple solution is to start with an empty database and then check every triple proposed for input. This solution assumes that an empty database exhibits semantic integrity with respect to any set of constraints. This is *not* true. Therefore, we have to impose a restriction on the type of constraints allowed so that we can assume that an empty database exhibits semantic integrity. The restriction is not severe and may be thought of as a point of clarification as far as the user is concerned:

The set of integrity constraints must not contain any constraints consisting of one literal with: no negation sign and with a variable-free triple. For example, constraints such as:

$$(e_{11}.\ \in.\ \text{companies})$$

are not allowed. This constraint states that e_{11} must be known to be a member of the set of companies at all times, therefore an empty database does not satisfy this. A database implementor wishing to make this constraint could do so by putting the triple (e_{11}. ∈. companies) into the database before it is handed over to the user community.

5.7 Efficiency considerations

It can be seen that the procedure *p-instant* could generate a large number of instantiations in some cases. Fortunately it is not always necessary to generate all of these explicitly:

(i) If an instantiation in which no further variable substitution can be made is found to be false then the whole process can terminate and the triple may be rejected.

(ii) If an intermediate instantiation is found to be false irrespective of further substitution (as described in Section 5.5) then the whole process can terminate and the triple may be rejected.

(iii) If an intermediate instantiation is found to be true irrespective of further substitution (as described in Section 5.5) then further substitution of that instantiation is not necessary and the *current* call of *p-instant* can terminate. An extreme example of this is where a triple TI matches a constraint C because it corresponds to a literal which does not contain a negation sign. In this case, all possible instantiations of C which are implied by TI must be true.

The revised version is:

proc *p-instant* (**triple** TI, TC, **constraint** C):

begin

substitute variable in TC by equivalent entities in TI, make the same substitution(s) throughout C;

if C is **false** irrespective of further substitutions
  then goto reject
  elsef C is **true** irrespective of further substitutions
    then skip

```
    else bool sub := false;
      for each triple TX in C still containing one or two
                                                  variables
            do begin
                [] triples TDX := retrieve (TX);
                if TDX isnt empty
                  then sub := true;
                       for each TD in TDX
                         do begin
                           p-instant (TD, TX, C)
                           end
                fi
            end;
      if sub = false
        then if C is false
                  then goto reject
             fi
      fi
fi
end;
```

goto *reject* causes the process to terminate and the triple to be rejected. A **goto** has been used for clarity and for efficiency. Whenever any instantiation is found to be false the whole process can terminate and there is no need for recursive 'ascent'. If the triple is not rejected, then it is inserted.

As illustration of the process, consider the following examples:

Take the example database in Section 5.4 and suppose that the constraints C2″ and C6″ hold:

C2″: $(X.\in.\text{students}) \vee \neg(X.\text{enrolled for.}\ Y)$

C6″: $(X.\text{ident.}\ Y) \vee \neg(X.\text{named.}\ N) \vee \neg(X.\text{lives in.}\ A) \vee \neg(Y.\text{named.}\ N) \vee \neg(Y.\text{lives in.}\ A)$

Example 1. Try to insert TI = $(e_8.\in.\text{students})$:

(i) Match C2″

(ii) Make the substitution $\{X := e_8\}$ giving:
$(e_8.\in.\text{students}) \vee \neg(e_8.\text{enrolled for.}\ Y)$

(iii) The literal $(e_8.\in.\text{students})$ is true irrespective of further substitution (since it contains the triple being proposed for insertion), therefore no further substitution is necessary. Therefore, we can assume that all instantiations of C2″ implied by TI are true, and TI may be inserted.

Example 2. Try to insert TI = $(e_3.\text{named. Peter})$:

(i) Match C6″

(ii) Make the substitutions $\{X := e_3, N := \text{Peter}\}$ giving:
$(e_3.\text{ident.}\ Y)$
$\vee \neg(e_3.\text{named. Peter}) \vee \neg(e_3.\text{lives in.}\ A)$
$\vee \neg(Y.\text{named. Peter}) \vee \neg(Y.\text{lives in.}\ A)$

(iii) Find the corresponding triple $(e_3.\text{lives in. London})$ in the database, and make the substitution $\{A := \text{London}\}$ giving:
$(e_3.\text{ident.}\ Y)$
$\vee \neg(e_3.\text{named. Peter}) \vee \neg(e_3.\text{lives in. London})$
$\vee \neg(Y.\text{named. Peter}) \vee \neg(Y.\text{lives in. London})$

(iv) Find the corresponding triple $(e_1.\text{named. Peter})$ and make the substitution $\{Y := e_1\}$ giving:
$(e_3.\text{ident.}\ e_1)$
$\vee \neg(e_3.\text{named. Peter}) \vee \neg(e_3.\text{lives in. London})$
$\vee \neg(e_1.\text{named. Peter}) \vee \neg(e_1.\text{lives in. London})$

(v) No more substitutions can be made, therefore this instantiation is evaluated and found to be false.

(iv) No more instantiations of C6″ need be generated and no more constraints need be matched. TI is rejected.

Example 3. Try to insert the triple TI = $(e_5.\text{named. James})$

(i) Match C6″

(ii) Make the substitutions $\{X := e_5, N := \text{James}\}$ giving:
$(e_5.\text{ident.}\ Y) \vee \neg(e_5.\text{named. James}) \vee \neg(e_5.\text{lives in.}\ A)$
$\vee \neg(Y_5.\text{named. James}) \vee \neg(Y.\text{lives in.}\ A)$

(iii) Try to find a triple corresponding to $(e_5.\text{lives in.}\ A)$ in the database, since there is no such triple, the literal $\neg(e_5.\text{lives in.}\ A)$ must be true, therefore further substitution of this instantiation is not necessary and *p-instant* can terminate its current call.

(iv) We have not finished yet, we must also test the instantiations resulting from the substitution $\{Y := e_5, N := \text{James}\}$. We will find that all such instantiations are also true.

(v) Since no more literals in C6″ correspond to TI, we can accept TI for insertion because it has not been rejected.

5.8 Implementation

A system based on these ideas has been written and implemented as part of a final-year undergraduate project.[24] The programs are written in S-ALGOL[29] and run on a PDP 11/44. The procedure for generating and evaluating instantiations implied by an incoming triple are coded in about 200 lines of S-ALGOL.

6. LIMITATIONS OF THE METHOD

The method described in this paper is only one step towards the automatic maintenance of database integrity with respect to any set of constraints. The method has limitations:

(i) Since SCHEMAL contains no constructs for dealing with sets, constraints such as C13 and C14 cannot be accommodated.

(ii) Although constraints such as C8 could be expressed in SCHEMAL as:
$[(X.\text{teaches.}\ Y) \wedge (Y.\in.\text{students})] \Leftarrow (X.\text{sets exam.}\ E)$
the method will not work. This rule would be transformed to two rules in clausal form:
$(X.\text{teaches.}\ Y) \Leftarrow (X.\text{sets exam.}\ E)$
$(Y.\in.\text{students}) \Leftarrow (X.\text{sets exam.}\ E)$
and the relationship between the two Ys is lost.

The reason that the method does not work is that the existentially quantified Y should be replaced by the Skolem function $f(x)$ when writing out the two new rules. This cannot be done in SCHEMAL since it lacks function symbols.

(iii) Constraints such as C10 could be expressed in SCHEMAL as:

$$[(Y.\ \text{ident. teaches}) \vee (Y.\ \text{ident. sets exam})] \Leftarrow [X. \in \text{lecturers}) \wedge (X.\ Y.\ Z)]$$

However, this is not in keeping with the philosophy behind SCHEMAL.

(iv) The method requires the whole database to be locked while integrity checks are being carried out for a single triple. Only when the triple has been rejected or accepted can another triple be considered for input. This situation can be improved somewhat since two triples could be tested simultaneously if the constraints which they matched formed disjoint sets. This improvement however, has not yet been implemented.

7. FUTURE DEVELOPMENTS

SCHEMAL is based on a subset of first-order predicate calculus but is simpler because it only accommodates binary relationships. However, SCHEMAL still inherits all of the limitations of first-order predicate calculus. One of the problems of moving to higher-order calculus is that consistency checking becomes increasingly more involved if not impossible. We, are currently working on a second version of SCHEMAL which is based on a simple subset of a higher-order logic. We shall continue to restrict the language to binary relations since this has not given us any problems yet and has made the task of maintaining database integrity quite straightforward.

At present, the method is limited to the insertion of single triples. We have not discussed the case where several triples have to be inserted simultaneously to preserve integrity. One approach to this problem would appear to be to *assume that all triples in the transaction are already in the database* and then check each one for semantic integrity as outlined in the previous sections. If any one triple is found not to comply with a constraint then *all* triples in the transaction are rejected.

Further work on the efficiency of the method is required: (i) the order of the literals in a constraint will affect the number of database retrievals and partial instantiations required. The effect of re-ordering constraints needs to be studied. (ii) the implementation currently existing assumes that the triples returned by a call of *retrieve* will fit into mainstore. This might not be true, in which case some form of 'lazy evaluation' of *retrieve* will be necessary.[30] A problem here is that lazy evaluation might require a disc access for each triple (as and when it is needed). However, such evaluation would save space since triples need not be copied into mainstore until they were required.

8. CONCLUDING COMMENTS

We have shown how semantic constraints which are expressed in a language called SCHEMAL may be used to maintain the integrity of a binary-relational database automatically. The method has been fully implemented as part of a final-year undergraduate project. The fact that it was possible to code and test the method in such a short time indicates the simplicity of this approach. This point has also been illustrated by the fact that the complete method has been described in a few pages. This is in marked contrast to most publications on database integrity which tend to involve lengthy descriptions of new constraint definition languages and very short (often absent) descriptions of the way in which integrity is maintained.

We recognize that SCHEMAL is limited in its expressive power and have given examples of constraints which cannot be accommodated by our method. However, we regard the design and implementation of this language as a useful step in the development of more powerful languages.

We also appreciate that SCHEMAL is not a particularly user-friendly language. One needs to know something of mathematical logic to use SCHEMAL properly. For example, it is quite easy to write a rule such as:

$$(X. \in \text{alive}) \Leftarrow \neg(X. \in \text{dead})$$

and expect the system to *reject* the triple (John. $\in$. alive) if the triple (John. $\in$. dead) is already in the database. In fact, the system would not reject this triple.

We are currently working on a high-level user-friendly language which would map into SCHEMAL. This language may categorize constraints *for user convenience*, but such classification would be ignored when the constraints are translated into SCHEMAL.

Acknowledgements

We would like thank Dr Ian Sommerville and Mr Bob Welham of Strathclyde University, and Dr Gray of Aberdeen University for their interest in this work.

REFERENCES

1. G. Wiederhold, *Database Design*, McGraw-Hill, New York (1977).
2. M. P. Klopprogge, Term: an approach to include the time dimension in the entity—relationship model, in *Entity-Relationship Approach to Information Modeling and Analysis*, ed. by P. P. Chen, pp. 477–512. Institute (1981).
3. J. R. Abrial, Data semantics, in *Data Management Systems*, ed. by J. W. Klimbie and K. L. Koffeman, North Holland, Amsterdam (1974).
4. J. J. Florentin, Consistency auditing of databases. *The Computer Journal* **17**, 52–58 (1974).
5. A. Blaser and H. Schmutz, Data base research: a survey. *Lecture Notes in Computer Science*, Vol. **39**, ed. by G. Goos and J. Hartmanis, Springer-Verlag, pp. 44–113. Berlin (1975).
6. K. P. Eswaran and D. D. Chamberlin, Functional specification of a subsystem for data base integrity, in *Proceedings of the International Conference on Very Large Data Bases*, MA, 22–24 (1975).

7. M. Stonebraker, Implementation of integrity constraints and views by query modification, in *Proceedings of the ACM SIGMOD Conference*, San Jose, California (1975).
8. M. M. Hammer and D. J. McLeod, Semantic integrity in a relational data base system, in *Proceedings of the International Conference on Very Large Data Bases*, Massachusetts (1975).
9. H. Biller and E. J. Neuhold, Semantics of data bases: the semantics of data models. *Information Systems* **3**, 11–30 (1978).
10. G. Pelagatti, D. Paolini and G. Bracchi, Mapping external views to a common data model. *Information Systems* **3**, 141–151 (1978).
11. N. Roussopulos, CSDL: a conceptual schema definition language for the design of data base applications. *IEEE Transactions on Software Engineering* **SE-5**, 481–496 (1979).
12. J. Mylopoulos, P. A. Bernstein and H. K. T. Wong, A language facility for designing database-intensive applications. *ACM Transactions on Database Systems* **5**, 185–207 (1980).
13. M. L. Brodie, The application of data types to database semantic integrity. *Information Systems* **5**, 287–296 (1980).
14. A. Borgida and H. K. T. Wong, Data models and data manipulation languages: complementary semantics and proof theory, in *Proceedings of the International Conference on Very Large Data Bases*, pp. 260–271 (1981).
15. D. W. Shipman, The functional data model and the data language DAPLEX. *ACM TODS* **6**, 140–173 (1981).
16. J. A. Robinson, A review of automatic theorem proving, in *Proccedings of Symposia in Applied Mathematics* **19**, pp. 1–18. American Mathematical Society, Providence, Rhode Island (1967).
17. N. Roussopoulos and J. Mylopoulos, Using semantic networks for database management. *Proceedings of the International Conference on Very Large Databases*, Boston, Massachusetts (1975).
18. J. Smith and D. C. P. Smith, Database abstractions: aggregation and generalization. *ACM Transactions on Database Systems* **2**, 105–133 (1977).
19. CCA, *ADAPLEX Reference Manual*, CCA, 575 Technology Square, Cambridge, Massachusetts 02139 (1982).
20. R. A. Frost, A. D. McGettrick and R. K. Welham, The simplified binary-relational view of the universe and a conceptual schema definition language based on it. *Technical Report*, University of Strathclyde, Glasgow (1981).
21. R. A. Kowalski, Logic for data description, in *Logic and Data Bases*, ed. by H. Gallaire and J. Minker, pp. 77–102. Plenum Press, New York (1978).
22. R. A. Frost, Binary-relational storage structures. *The Computer Journal* **25**, 358–367 (1982).
23. J. Asher, The design and implementation of a conceptual schema definition language, Final Year Project, Department of Computer Science, University of Strathclyde (1982).
24. S. Whittaker, Implementation of an automatic semantic integrity maintainer. Final Year Project, Department of Computer Science, University of Strathclyde (1982).
25. N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York (1971).
26. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey (1976).
27. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin (1981).
28. J. Minker, in *Logic and Data Bases*, ed. by H. Gallaire and J. Minker, Plenum Press, New York (1978).
29. R. Morrison, *S-Algol Reference Manual*, Department of Computational Science, University of St Andrews, Scotland (June 1980).
30. P. Henderson, *Functional Programming*, Prentice-Hall, Englewood Cliffs, New Jersey (1980).

Received April 1982