Integrity aspects of a shared data base

by EDUARDO B. FERNANDEZ and RITA C. SUMMERS
IBM Los Angeles Scientific Center
Los Angeles, California

ABSTRACT
A simple model is formulated to represent integrity constraints and to describe the evaluation and enforcement of these constraints in a shared data base environment. The proposed model is applied to define the integrity facilities of a relational data base. This data base consists of a set of base relations, which are presented to the user through application-oriented views. These views are basically joins or projections of the base relations, and the data base is accessed through a language which by referring to predefined views, makes explicit the intentions of application programs. The types of integrity rules incorporated in this system, their description, their evaluation, and the propagation of changes to views into the shared data base, are discussed.

INTRODUCTION
Information protection, i.e., the prevention of illegal disclosure, modification, or destruction, and of invalid modification, of the contents of the data base, is one of the most critical problems of data base systems. For this effect, the data base should include an authorization system to prevent illegal access to the information, and an integrity system to prevent some types of inconsistencies, introduced by errors of the users or their application programs. By enforcing semantic restrictions on the information, it is possible to insure that the contents of the data base are at least plausible, if not correct, and that no inconsistencies exist between related information.

An integrity system for a data base consists of a set of assertions about the contents of the data base (expressed in some suitable language), a validation mechanism that checks changes to the data base for compliance with the integrity assertions, and an enforcement mechanism that performs some predefined actions upon detecting that one or more of the integrity assertions have been violated. In those systems where users have access to a shared data base through views,6,17 the integrity system should also contain specifications about how changes are reflected to the shared data.13

Typical integrity assertions are statements such as: “Salaries must be positive.” “Salaries must be non-decreasing.” “Student numbers must be present in enrollment lists.” Present commercial systems have only basic capabilities to define and enforce these types of statements,9 and more advanced integrity checking is left to the application programs. In large shared data bases, it is clear that preserving integrity should be a system function, not left to the individual applications.

A model for the functions of an integrity system is presented in this paper (second section), which describes the structure of the integrity constraints, their evaluation, and their enforcement. This model is applied in the following sections to define an integrity system for a shared relational data base described in previous papers.5,16 The third section contains a brief description of this data base, the type of integrity assertions included in it, the association of these assertions with data objects, the way of describing the assertions, and their evaluation. The use of compile-time actions to prepare integrity checking, and the effect of changes to the data base through views, are considered. The fourth section compares this approach to other proposals, while a final section provides some conclusions.

A MODEL FOR INTEGRITY
A simple model for integrity is now presented, which can accommodate most of the features found in recent discussions of integrity characteristics.1,4,6,9,12,13,17 It is described in terms of a relational model of data,6 and it is used to guide the selection of integrity features for a relational data base. However, a substantial part of the subsequent development could be applied to other logical models of data as well.

Integrity rules
An integrity rule is the 5-tuple \((o_t, a_t, p_c, e_m)\), where \(o_t\) is the data object to which the rule applies, \(a_t\) is an access type which indicates for what type of data base access the rule will be invoked, \(p_c\) is an assertion
stating a semantic constraint which must be true for an occurrence of the object \( e_0 \), \( e_c \) is a predicate expressing a condition which must be true in order for \( p_k \) to apply to \( e_0 \) and \( e_m \) is an enforcement type that specifies the action that will be taken by the system if \( p_k \) is not true.

For specific systems it may be convenient not to separate the five components of the rules; for example \( t_k \) and \( c_k \) could be combined to specify a condition for application of \( p_k \) or \( c_k \) could be part of \( p_k \). Also, in most cases the \( t_k \)'s are a small and predefined set, and the specification for \( t_k \) need not be explicit. However, it is important for conceptual clarity in the design of the integrity system, to separate these components as independent units.

As an illustration let us consider a student data base where record type or relation STUDENT contains the fields NAME, ADDRESS, COURSE, and GRADE. An integrity rule for object STUDENT. GRADE could contain an assertion \( p_k \) such as “Grades must be ‘A’ or ‘B’ or ‘C’ or ‘D’ or ‘F’.” This assertion would be enforced whenever some access actions involving field GRADE, specified by \( t_k \), are performed; for example if \( t_k \) indicates “update, insert,” then validation of new values for field GRADE will be performed. However, if a condition \( c_k \) is specified, for example: “When COURSE \( \neq ‘123A’ \)” the validation test will only be applied for those tuples where the COURSE field value is not ‘123A’. An enforcement type specification could indicate “log and record invalid tuple(s) in INVALID,” indicating that on violation of the assertion the name of the program, time of day, etc., will be logged and the tuple or tuples not satisfying the assertion will be stored in a special table called INVALID.

In a system containing many interobject constraints, each of them involving several objects, a more convenient definition for an integrity rule would replace object \( o_i \) by a set of objects to which the assertion applies. However, most of the interobject constraints found in practical systems involve only two data objects, such as “Enrolled students must be registered students.” In this case, it suffices with splitting the constraint into two constraints of the form indicated earlier, and associating each constraint with each object. In the example, we obtain two assertions: “The set of enrolled students is included in the set of registered students,” associated with object “enrolled students,” and “The set of registered students includes the set of enrolled students,” associated with object “registered students.”

In a relational system the objects \( o_i \) are either domains or relations. In those data base systems which do not have domains as separate entities, domain constraints are replaced by constraints associated with field types, where a field type is a prototype for fields that carries a set of attributes that apply to all the fields based on the corresponding type. The integrity constraints associated with a given type apply similarly to all the fields based on the type. Relation constraints include interrelation between fields or constraints that apply to specific fields in the relation.

There is sometimes a rather fine line between an integrity rule and an access rule. An access rule was described in References 11 and 19 as a tuple \((S, o_i, t_k, p_k, e_m)\), where \( S \) is a subject or requesting entity, and all the other terms are the same as in integrity rules but with a different interpretation. There, \( o_i \) is the object accessible to \( S \); the type of access authorized to \( S \) for \( o_i \) is given by \( t_k \), and it is only granted if predicted \( p_k \) is true. Finally, \( e_m \) specifies an action to be performed by the system if an illegal access is attempted. As either access or integrity rules include predicates, application of an integrity rule can depend on the identity of the user, and an access rule can depend on data values. However, access can be decided in some cases by looking only at the names of the requested objects, while integrity control depends always on the contents of the objects involved. Further, the access types \( t_k \) specified in an integrity rule always refer to access that modifies the data base, while access control refers to any type of access, including just inspection of the contents of some data unit. Notice also that access rules do not include conditions for their application, but every access must be validated by the system.

**Validation of integrity rules**

Integrity assertions can be classified into a few basic categories\(^\text{10,11,12}\) which correspond to specifications of range, sets of permitted values, format, uniqueness of some value, non-missing values for a field, new vs. old values (transition assertions in Reference 10), and interfield assertions. Validity tests for these categories can be symbolically described as follows.

(a) Update validity test.

Let \( v \) be a new value for a field \( f_i \) corresponding to domain or type \( F_{n_i} \), which is part of relation \( R_i \). Then, the validity condition is

\[
\text{is-valid} (v, f_i, R_i) \iff 
\begin{align*}
&\text{for } P = (p_k \land (o_i = F_{n_i} \lor R_i) \land (t_k = \text{‘update’}) \\
&\land (e_m = \text{‘true’}), \\
&\text{P (v) = ‘true’.}
\end{align*}
\]

This test establishes that all the predicates of the integrity rules that apply to object \( o_i \) and for which their condition predicates are true must be satisfied, in order for the new tuple including the changed value \( v \) to be acceptable. The object \( o_i \) is taken to be either the field type for \( f_i \) or the relation including \( f_i \), since either of these can contain assertions relevant to \( f_i \).

(b) Insertion/deletion validity test.

Let \((v_1, v_2, \ldots, v_r)\) be a tuple to be inserted into or to be deleted from relation \( R_n \), and that specifies values
for fields $f_1, f_2, \ldots, f_n$, respectively, corresponding to
domains $F_1, F_2, \ldots, F_n$, respectively.

Now the validity condition is

$\text{is-valid} \left( (v_1, v_2, \ldots, v_k), (f_1, f_2, \ldots, f_n), R \right) \Rightarrow$

for $P_i = \left( p_{i|} (a_i = F_i \lor R_i) \land (t_i = \text{'insert'} \ (\text{delete})) \right) \land (c_i = \text{'true'})$

$P_i (v_i) = \text{'true'}, \ 1 \leq i \leq k$.

These validation tests assume that the assertions are
associated with all the relevant fields. For example an
assertion "Field $f_i$ must be a subset of field $f_j$" would
also imply the assertion "$f_j$ includes $f_i$".

When users have access to the shared data base
through application-oriented views, it is necessary
to have special reflection rules which specify how
changes to the views are reflected to the shared data
base. The resultant tuples, which are tuples of the
base relations, must then be validated as above.

A reflection rule has the general structure $(v_i, a_i, t_k)$,
where $v_i$ is the view to which the rule applies, $a_i$ is a
data object, and $t_k$ an access type (which for fields
can be 'null' or 'update', and for relations can be
'null', 'insert', or 'delete'). A rule of this type specifies
that for view $v_i$, a change of type $t_k$ to object $a_i$ in the
view results in a corresponding change on $a_i$ in the
shared data base.

The following example illustrates the reflection rule
mechanism. Let $v_i (f_1, f_2, f_3, f_4, f_5)$ be a view including
fields $f_1, f_2, f_3$, and formed by the join of relations
$R_1$, $R_2$, and $R_3$, such that $f_1, f_2 \in R_1$, $f_3, f_4 \in R_2$, $f_5 \in R_3$ (that is, $f_1$ and $f_2$ are the joining fields). Let the
following reflection rules be defined $(v_i, f_6, \text{update})$,
$(v_i, f_7, \text{update})$, $(v_i, f_8, \text{insert})$, $(v_i, f_9, \text{insert})$. Updates through $v_i$ are then reflected as follows:
update $f_6$ in $R_1$, update $f_7$ in $R_2$, and $R_3$ update
in $R_3$. Integrity constraints referring to the fields
of the base relations can now be applied. Insertions
through $v_i$ are reflected as: insert tuple in $R_1$, insert
tuple in $R_2$, subject to any integrity constraints affecting
$R_1$ and $R_2$). Notice that if we had an additional
reflection rule such as $(v_i, f_7, \text{update})$, then updates to
both $f_6$ and $f_7$ are effectively insertions into $R_3$, and
must be reflected as such. Also if the views are formed
with projections of some of the relations, then some of
the reflected tuples will have unspecified fields, and
may thus violate integrity constraints that prescribe
specified values for given fields.

FUNCTIONAL SPECIFICATIONS FOR AN
INTEGRITY SYSTEM FOR A
SHARED DATA BASE

The model of the previous section includes most of
the features which are of importance in a practical
system. It is used in this section to define an integrity
system for a shared data base proposed in previous
papers. The data description language is used
to present the functional capabilities of the system. A
mechanism to support these functions is given in the
fourth section of this paper.

The data base system

This data base consists of a set of base relations (or
Data Base Structures), which are presented to the
data base. Views, or templates, are basically joins and/or
operations of the base relations, constructed for specific
purposes. The data base is accessed through an extended
high-level language, which, by referring to predefined
templates, allows the user programs to manipulate
data base elements as any other program variable.
The organization of this data base lends itself very
well to apply to it an authorization model, which,
due to the fact that the extended language makes
program data intentions manifest, can be enforced
partially at compile time. Even those access decisions
which are data-dependent and cannot be enforced at
compile time can be prepared at this time, thus
decreasing validation overhead at execution time.

The model of data described in Reference 18 uses
four kinds of data objects: templates, data base structures,
fields, and types. (The term type replaces the
term field type used in earlier papers.) The DB structures constitute the shared system view of the data
base, and they are composed of one or more fields.
Types, which carry attributes, serve as prototypes for
fields. A template is an aggregate (typically a join)
of DB structures that is constructed for a specific
purpose. A template includes a defining expression,
which indicates which DB structures participate in its
construction and how they are combined. An application
program views the data base as a set of templates.
The terms DB structure and template will be
used in two senses: as structural entities or definitions,
and as sets of occurrences of those entities.

Four basic roles for users are contemplated in this
system—data base administrators (DBA) define and
maintain the shared DB structures and their integrity
and authorization rules; data base designers build
templates to perform specific functions; application
programmers use the predefined templates to write
application programs; application users invoke their
authorized programs to perform specific actions on
the data base.

A data description language for this data base has
been proposed in Reference 19, where its functions,
use, and syntax are presented in some detail. As its
syntax is simple and almost self-explanatory we shall
use this language in our examples without further
explanations. We are not concerned here with syntac­
tical details and the use of this language is purely to
illustrate the functional capabilities of the system.
Integrity rules

Integrity rules can be associated with both types and DB structures. They can be given as part of the object declaration or as independent rules. A rule associated with a type automatically applies to all fields based on that type. The goal is to simplify the descriptive tasks—if a rule is specified for a type, it need not be written for all the fields based on that type. DB-structure rules, enhanced by the relevant type rules, are the basic integrity rules for the shared data base. In addition, the template's defining expression provides criteria for new occurrences created through that template.

An integrity rule can contain references to either built-in functions or procedures provided by an installation. These latter are data base procedures in the sense of the CODASYL DDL report, or formulary procedures, and should be carefully certified. Data base procedures also play a role in the enforcement of integrity and security. Both integrity rules and access rules include an enforcement type, which specifies the name of another system object called an action. Some action names are built-in (for example LOG, NOTIFY_SECURITY_OFFICER); other actions are defined by ACTION declarations, which can specify procedure names.

The following are examples of type declarations including integrity assertions:

```
DECLARE EMPNO TYPE WHERE (*>10000 & *<100000); DECLARE STATUS_CODE TYPE FIXED BINARY WHERE (*<100);
```

The declarations for EMPNO and STATUS_CODE specify ranges of admissible values. (The asterisk is used in place of the type or component name.)

The declaration of a DB structure specifies the fields of the structure (by reference to types) and integrity rules. The following are examples of DB-structure declarations:

```
DECLARE 1 SUPPLIER DBSTRUCTURE
  WHERE(NAME!=MISSING),
  2 NUMBER LIKE SUPPLIER_NUMBER,
  2 NAME LIKE BUSINESS_NAME,
  2 STATUS LIKE STATUS_CODE,
  2 CITY;

DECLARE 1 PART DBSTRUCTURE WHERE
  (NUMBER!=MISSING),
  2 NUMBER LIKE PART_NUMBER,
  2 NAME,
  2 COLOR,
  2 WEIGHT;

DECLARE 1 SP DBSTRUCTURE,
  2 SPKEY,
  3 S_NO LIKE SUPPLIER_NUMBER,
  3 P_NO LIKE PART_NUMBER
  2 QUANTITY;
```

SUPPLIER is a DB structure with four fields. The value of SUPPLIER.NUMBER is required; that is, it cannot be undefined (denoted by MISSING in this particular syntax). LIKE indicates correspondence between fields and types, i.e., NUMBER must obey the integrity rules of the type SUPPLIER_NUMBER, CITY of the CITY type, etc. Uniqueness requirements, and integrity rules that involve DB structures other than the object of the rule, are discussed in the fourth section of this paper.

The previous examples expressed integrity assertions as part of the object's declaration. If there is more than one rule for an object, if the rule is complex, or if more dynamic integrity assertions are required, separate statements can be used. The following rule, for example, states that an attempt to set STUDENT.Name to MISSING causes the entire occurrence of STUDENT to be deleted. DELETE is a built-in action.

```
STUDENT IS CONSISTENT WHERE(NAME!=MISSING) ENFORCE(DELETE);
```

The following rule states the requirement of no A's in a specific course:

```
ENROLLMENT IS CONSISTENT WHERE
  (GRADE;='A') ENFORCE(LOG)
  WHEN(CN='CS123A');
```

The WHEN expression is evaluated first and determines if the WHERE expression is to be evaluated. Unqualified names on the left side of comparisons in the expressions are implicitly qualified by the object name. Reference is to the new value of the occurrence, unless the old value is explicitly specified, as in

```
WHERE(COUNT>OLDVALUE(COUNT))
```

Integrity expressions may be specified in three ways: WHERE, WHERE SOME, and WHERE NONE. The meaning of each form is given in a later section. SOME can be used, for example, to express the requirement that CITY must be an element of CITYVAL, as in

```
CITY IS CONSISTENT WHERE SOME
  (CITY=CITYVAL);
```

To require that PART.NAME be unique, we write,

```
PART IS CONSISTENT
  WHERE NONE(NAME=OLDVALUE(NAME));
```

However, this last constraint is awkward, and could be simplified by introducing the concept of uniqueness in the DDL (see for example Reference 10).

Actions

The concept of an action provides a generalized way of making events contingent on the state of the data base system. Logging, report generation and various
periodic activities are declared as actions. Logging of
data accesses is a valuable security tool if the content
of the log can be dynamically controlled and if access
to the log itself is also controlled. (We refer here to
selective logging as opposed to complete journaling
that is done for recovery purposes.) Actions also pro-
vide a way to start and stop the gathering of statistics
or performance measurements.

An action is specified as either (1) a procedure in-
vocation, or (2) combinations of other actions, de-
clared or built-in. It can have initiating conditions
specified in the form of a WHEN expression, or can be
initiated as enforcement of an access or integrity
rule. Built-in actions execute with the rights of the
system. Administrator-defined actions execute with
the rights of the DBA application and the user class of
the definer. The WHEN expression can refer to sys-
tem data only. For example, to initiate an action when-
ever a new part is added to the data base

DECLARE NEW PART ACTION(LOG,PROC
(P1)
WHEN (REQUEST.OBJECT=‘PART’ &
REQUEST.ACCESS=‘ALLOCATE’);

In the example above, LOG is a built-in ACTION.
P1 is a procedure declared by the administrator, and
REQUEST is a system DB structure. The REQUEST
and SYSTEM data values are available to the pro-
cedure; it can use any other data base items accessible
to its application and user class.

There exist some problems associated with the use of
actions, not all of which have satisfactory solutions
until now. Some of these problems are:

(i) integrity violations within actions, which could
result in a never-ending sequence of actions invoking
actions;

(ii) name resolution at the moment of invocation of
the procedures in the actions;

(iii) access rights of the invoked actions.

A simple, although somewhat unsophisticated solution
for (iii) is given in this section. Problem (ii) is dis-
cussed later.

Updates through views

From the viewpoint of an application program, all
modification of the data base occurs through templates.
From the viewpoint of the system as a whole, all modi-
fication is effected through changes to DB structures.
One of the functions of the view mechanism is to pro-
vide the transition from a template-expressed change
to an unambiguous DB-structure change. Many prob-
lems arise in this context, not all of which have been
solved. It is clear, however, that the system’s basic
integrity rules apply to DB structures. The template
definition can place additional constraints on changes
made through that template.

A template, like a DB structure, is a structure that
can also be viewed as a relation or table. A template
can join two or more DB structures, eliminate rows
and columns of the resulting table, and permute and
rename columns. For example, using the data base
definitions, the following template joins SUPPLIER
and PART, using SP as intermediate. It also deletes
the fields NUMBER and STATUS from SUPPLIER
and all fields but NAME from PART, renames CITY to
PLACE, and reorders the remaining fields.

DECLARE 1 PARTSLOC TEMPLATE
WHERE (SUPPLIER.NUMBER=SP.S_NO
& SP.P_NO=PART.NUMBER),
2 SUPPLIER,
3 PLACE LIKE CITY,
3 NAME,
2 PART,
3 NAME;

The WHERE expression could have been extended by,
for example,

& SUPPLIER.CITY='LONDON'

to eliminate rows or occurrences. If a template in-
volves only a single DB structure, and does not select
rows, no WHERE expression is needed.

If two fields compared in the WHERE expression
are not based on the same type, the template designer
receives a warning; if the two cannot legally be com-
pared (according to PL/1 rules), the template is re-
jected. When two fields with unlike attributes are com-
pared, conversion takes place. Attributes may appear
on field names in the template; if they do, conversion
from or to the attributes of the type will occur when
data is accessed.

The template has a visible aspect (the portion that
appears in the declaration below the defining expres-
sion) and a hidden aspect (all other fields of the par-
ticipating DB structures). A program can use only
the visible aspect; access and integrity rules can also
refer to hidden fields. The user who defines a template
must have READ NAME access to all objects whose
names appear in the template.11

To summarize the definition of a template, it is a
projection (not purged of duplicates) and permuta-
tion of a subset of the cartesian product of the DB
structures named in the defining expression and in the
visible aspect. The subset is specified by the defining
expression, the projection by the fields appearing in
the visible aspect, and the permutation by the order in
which the fields appear.

Each template represents a specific intent with re-
spect to data base access. Not all templates are in-
tended to be used for changing the data base, for ex-
ample. Those that are have attributes that limit the
kinds of changes that can be made through the tem-
plate and that resolve any ambiguities about how the change is made.

The UPDATE attribute can appear on template fields. ALLOCATE and FREE are attributes of the template. Allocation of a template by a program, following by setting of field value means: "ensure that an occurrence with this value exists in the database". This could possibly require adding occurrences to more than one DB structure. The ALLOCATE attribute can limit which DB structures will be affected, as in the following attribute for the PARTSLOC template

ALLOCATE(SP)

which does not allow occurrences of SUPPLIER or PART to be allocated, but only of SP. If there existed no occurrence of SUPPLIER with the NAME value of the new template occurrence, the ALLOCATE would be rejected and an error condition raised. A template with no ALLOCATE attribute cannot be used for allocating. The FREE attribute allows FREE operations to be performed on a template and resolves any ambiguities about which DB-structure occurrences are to be deleted. For example, the program statement

FREE PARTSLOC;

could be implemented by deleting occurrences of any of three DB structures. The template attribute

FREE (SP)

specifies which DB structure is intended.

The UPDATE attribute specifies that a template or a template field can be updated, and may also specify propagation of the update to other (possibly hidden) fields of the template. For example, the following template, which is used for changing a part number, updates PART (its source DB structure) and also SP.P_NO, which is compared for equality with PART.NUMBER in the defining expression.

DECLARE 1 CHGPN TEMPLATE WHERE
(PART.NUMBER=SP.P.NO),
2 PART,
3 NAME,
3 NUMBER UPDATE(EQUAL);

The template's defining expression acts as an integrity rule, since all updates made through the template must satisfy the expression. (Updates need not satisfy the application program selection expression, however).

The UPDATE, ALLOCATE, and FREE attributes specify what can be done through a template, regardless of application or user class, i.e., they implicitly define reflection rules. Access rules further constrain use of the template in specific contexts. A more general mechanism to specify reflection rules could be constructed by the use of actions.

AN INTEGRITY MECHANISM

General aspects

It is clear that a mechanism to perform validation and enforcement of the integrity assertions discussed earlier will involve a good increase in overhead with respect to a system not providing these functions. It is then important to define a mechanism able to support the desired functions at the lowest possible cost. The cost function to be minimized will be the amount of CPU time involved in validating the assertions, and it will be shown that the data base architecture under consideration provides a convenient environment for efficient evaluation of integrity constraints. As it has been pointed out by several authors, the way in which the data base is structured has a significant effect on the validation effort as well as on the type of inconsistencies that can occur. However, neither the model described here nor this data base have anything special to offer in this respect, and the performance implications of data base structuring will not be considered.

Evaluation and enforcement of integrity rules

As indicated by Florentin, a basic condition to have an effective validation system is a user interface consisting of preformatted transactions. This is the case in this data base, where all manipulations on the data base are made through predefined templates. The pre-specified structure of the template makes it possible to validate integrity assertions in a systematic and disciplined way.

Any change to the data base must satisfy both the template's defining expression and all integrity rules for all the DB structures that are changed. The defining expression is first evaluated on the new template occurrence, and the change is rejected if the expression is false.

Integrity rules associated with a DB structure can involve fields in other DB structures. For example, values of one field may be required to be a subset of the values of another. The concept of an integrity template is introduced to define how these rules are evaluated when the DB structure is updated. There are three interpretations of the integrity template, corresponding to WHERE, WHERE SOME, and WHERE NONE. Assume a new or changed occurrence of a DB structure DBI, with a new value of dbi. For each integrity rule that has DBI as an object, we consider a WHEN-template whose defining expression is (DBI= dbi & when-expr). If the WHEN-template is empty, the rule is ignored. For each category of WHERE, we consider an integrity template that is the cartesian product of all DB structures appearing in the effective integrity expression (which is the AND of the remaining rules), selected for (DBI= dbi). Then, depending
on the category, we require that the effective integrity expression be:

1. true for all occurrences (WHERE)
2. true for at least one occurrence (SOME), or
3. true for no occurrence (NONE) of the integrity template.

Changed occurrences become a part of the shared database when unlocking occurs. During the interval between update and unlocking, violations of certain types of integrity rules may exist, but these are seen only by the process making the changes.

As the data manipulation language used in this system makes explicit the data actions of application programs, it is possible to determine at compile time exactly what the program is intending to do with respect to the database. This allows to generate code at this time to evaluate and to enforce access rules. In a similar way, code can be generated to evaluate integrity rules. In effect, by looking at the type of access of the program with respect to a given data object, it is possible to decide by looking at the $t_o$ of the integrity rules for $o_o$, which of these rules apply to the given program. The conditions for the relevant rules will normally only be able to be evaluated at execution time since they are usually data dependent, and the same is true for the predicates $p_o$. However, as the data objects which have to be retrieved to evaluate these predicates are known at compile time, code can be generated for their efficient access. At the same time, code to perform any enforcement action can be generated. Problems with name resolution in the procedures that can be present in $p_o$, $c_o$, or $e_o$, can be decided if compile-time actions include flow of control analysis. Naturally, compiled programs become sensitive to changes in the relevant integrity rules, which result in recompilation of all the programs where they are used.

All this early processing permits considerable reduction in the overhead necessary to perform access control and integrity checking at execution time. For systems where even this overhead is excessive, integrity rules can be evaluated periodically rather than at every modification of the data object. This can be done easily in this system by defining an action to be an integrity check, and giving that action an appropriate WHEN condition.

Example,

```
DECLARE A1 ACTION (INTEGRITY(PART))
  WHEN (SYSTEM.TIME='0000') :
```

As in this system file descriptions are external to application programs, it is possible to use as a mechanism for enforcement a strategy similar to the one proposed in Reference 8, where “surveillance routines” are attached to files where given conditions should be enforced. The only difference is, that instead of these routines being attached by the execution supervisor to the application programs accessing the file, they are attached at compile time.

### DBA facilities

A fundamental requirement of an integrity system is to provide a convenient interface for the data base administrator, to permit an easy, consistent, and efficient way of defining and maintaining integrity rules. The data description language presented here in distributed form has been designed to be simple and complete. The integrity rules are themselves stored together with all other data definitions, access rules, action declarations, etc. using the same mechanism as the data files of the shared database. In other words, integrity rules and all other data descriptions are data base structures and constitute a “data base about the data base,” which incorporates the information usually associated with data dictionaries/directories. Some of the integrity rules therefore, could refer to the consistency of the relations storing integrity rules. Also, information about which programs are affected by changes to a given integrity rule should be part of this facility.

### RELATED WORK

The separation between the logical and physical aspects of a data base makes it possible to define high-level integrity systems which can be analyzed for consistency and completeness. This has resulted in several approaches to integrity. It is then important to compare our results to these to put things in perspective.

With respect to integrity, the CODASYL DDL provides the CHECK clause, which can specify either ranges of values, expressed in literals, or the name of a data base procedure. The value range is only one of many types of required integrity constraints; and the procedure invocation mechanism allows integrity requirements to be buried in procedural code, as they are buried in application code in today’s data base systems. Such a mechanism has to be provided, but as a last resort after more explicit ways of expressing integrity have been exhausted.

A comprehensive treatment of integrity is given by Stonebraker in describing the INGRES system, in which an integrity assertion is stated as one or more range statements, plus an integrity qualification. The range statements define a cartesian product, and the qualification is true or false for each tuple in that product. This is equivalent to our integrity template, but the lack of a WHEN specification causes rather awkward qualifications. For example, the rule that everyone in the toy department must make more than $8000 is expressed in INGRES as

```
RANGE OF E IS EMPLOYEE
INTEGRITY E.SALARY>$8000 or E.DEPT='toy'
```

From the collection of the Computer History Museum (www.computerhistory.org)
as opposed to our

```
EMPLOYEE IS CONSISTENT WHERE
(SALARY>8000)
ENFORCE WHEN (DEPT='TOY');
```

Another important relational data base is the SEQUEL system, whose integrity aspects are discussed in References 2 and 10. The functional specifications of the integrity subsystem of SEQUEL are in general consistent with the approach presented here. The concept of condition in their integrity rules is imbedded in the assertions. The concept of action is present in their "failure actions," which define how the system responds to integrity violations. Update through views is discussed separately from integrity. The view concept of the SEQUEL system allows the subschema to define joins (as well as other structures) and to convert between units (such as dollars and lire). The template differs from the SEQUEL view primarily in the handling of changes to the data base. A template is the concrete representation of a specific intent regarding use of the data base. Rather than being defined by the casual user of a query language, it is designed by a professional application designer and installed by a DBA. Therefore, rather than applying the "uniqueness rule" of Reference 3 (that a change to a view is permitted only if there is a unique change to the underlying base relations that will result in the view change), we allow the template's definition to choose one of several possible changes, or to disallow a unique change. In any case, it is clear that a system like SEQUEL is compatible with this latter approach, i.e., it would be possible to have data base administrators and application designers building views for casual users.

Florencin has studied integrity from a more theoretical point of view, stressing the value of predicate calculus in defining and evaluating integrity conditions. However, his cost function for the calculation of integrity constraints is based on the number of sequential file searches instead of minimization of CPU time as in our case. Graves has given a very complete discussion of the functions needed in any data description language for integrity purposes. Their concept of condition, and the delayed assertions of Reference 10 are found in his specifications. The syntax of his DDL is COBOL-oriented while our syntax is PL/I-oriented, however most of his concepts are consistent with our approach. He was also the first one considering update effects an integrity problem. Hammer and McLeod have provided a detailed discussion of the nature of integrity constraints; however, they do not consider the problem of updates through views or the evaluation of the constraints. Their work is of great value to define the descriptive aspects of integrity constraints and the functional capabilities of the supporting system.

CONCLUSIONS

A model of the functions of an integrity system for a shared data base has been proposed. Such a model is valuable to guide the design of the functional specifications of integrity systems for specific data bases. The model includes not only the characterization of integrity constraints but also the handling of updates to the data base through views, which is considered here as an important aspect for preserving the semantic integrity of the information stored in the data base. The model puts together different aspects of integrity, which until now have only been partially present in specific proposals.

A relational shared data base presented in earlier papers is then shown to represent a very convenient embodiment of the integrity model. The particular characteristics of this data base, i.e., use of an extended high level language for data manipulation, direct reference to the data base variables, view interface for end users, use of compile-time actions, provide an environment in which the definition, evaluation, and enforcement of integrity constraints can be performed with relatively low overhead, with respect to systems incorporating similar functions.

REFERENCES

13. Graves, R. W., "Integrity Control in a Relational Data