*Malaysian Journal of Computer Science, Vol. 14 No. 2, December 2001, pp. 70-80*

# DERIVING GLOBAL AND LOCAL INTEGRITY RULES FOR A DISTRIBUTED DATABASE

***Hamidah Ibrahim***
Department of Computer Science
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
43400 UPM Serdang
Tel.: 03-89486101 ext. 6510
email: hamidah@fsktm.upm.edu.my

***ABSTRACT***

*An important aim of a database system is to guarantee database consistency, which means that the data contained in a database is both accurate and valid. Integrity constraints represent knowledge about data with which a database must be consistent. The process of checking constraints to ensure that update operations or transactions which alter the database will preserve its consistency has proved to be extremely difficult to implement, particularly in a distributed database. In this paper, we describe an enforcement algorithm based on the rule mechanisms for a distributed database which aims at minimising the amount of data that has to be accessed or transferred across the underlying network by maintaining the consistency of the database at a single site, i.e. at the site where the update is to be performed. Our technique referred to as the integrity test generation, derives global and local integrity rules has effectively reduced the cost of constraint checking in a distributed environment.*

***Keywords:*** ***Distributed Database, Integrity Constraints, Integrity Constraint Enforcement***

## 1.0 INTRODUCTION

Guaranteeing *database consistency* has been an important issue in centralised databases over the last decade and, not surprisingly, much attention has been paid to the maintenance of integrity in these systems. Although this research effort has yielded fruitful results that have given centralised systems a substantial level of reliability and robustness with respect to the integrity of their data, today's DBMS technology still offers limited support for the automated verification of constraint satisfaction and enforcement. The crucial problem is the difficulty of devising an efficient generalised algorithm for enforcing database integrity against updates. The growing complexity of modern database applications plus the need to support multiple users has further increased the need for a powerful and efficient enforcement strategy to be incorporated into these systems. Most of the approaches proposed for improving the efficiency of constraint checking are not suited to a distributed database environment. Moreover, devising an efficient algorithm for enforcing database integrity against updates is extremely difficult to implement and can lead to prohibitive processing costs in a distributed environment [8, 9].

In the literature, three approaches for guaranteeing database consistency have been reported. In the first approach, the responsibility for ensuring the consistency of the database when a transaction occurs is part of the transaction design process. The transaction designers are responsible for ensuring that transactions are *safe*: i.e. when executed, the transactions are guaranteed to bring the database from one consistent state to another. Consequently, transactions can get very complex and a *transaction design tool* is usually incorporated into the system to assist designers to construct safe transactions. In the second approach, transactions have integrity tests embedded in them to perform the necessary integrity checking. The modified transactions can then be executed by standard transaction facilities. This approach is based on the *query modification*[1] and *transaction modification* strategies, where an arbitrary query or transaction that may violate the integrity of a database is modified, such that the execution of the modified query or transaction is sure to leave the database in a consistent state. This is the case in the SABRE [9] and the PRISMA [2] projects. In the third approach, integrity tests are general rather than transaction-specific and thus, no knowledge of the internal structure of a transaction is required. Typically, this approach requires rule mechanisms to implement integrity constraint enforcement. This approach is employed by the Starburst project [1] and the latest versions of commercial DBMSs such as INGRES and ORACLE.

[1] The term *query* here denotes a request that requires changes to the database state.

In [4], we introduced an integrity constraint subsystem for a distributed database (SICSDD) that we have developed. By database distribution, we mean that a collection of data which belongs logically to the same system is physically spread over the sites (nodes) of a computer network where inter-site data communication is a critical factor affecting the system's performance. The integrity enforcement utilized by the SICSDD subsystem is based on the rule mechanism. This approach is adopted because:

(i) the integrity tests employed are not fixed at compile time and thus, tests can be selected from alternatives according to conditions in the database at run time – this flexibility leads to greater efficiency which is desirable for a distributed database;

(ii) logical independence between update operations and integrity constraints is supported, as the binding between them is deferred until an update is submitted – this leads to better constraint optimization particularly for distributed constraints; and

(iii) the users no longer have to worry about consistency preservation, as this is supported by the subsystem. Integrity in the SICSDD subsystem is maintained by fully exploiting the available information at a target site. In our work, knowledge about the database application is exploited to (i) derive a set of simplified constraints that can be straightforwardly used for constructing efficient enforcement algorithms, and (ii) infer the information stored at different sites and so minimize the support from remote sites required during the evaluation of these constraints.

This paper proposes an algorithm for deriving efficient distributed integrity rules for maintaining semantic integrity in a distributed database. The main property of the proposed algorithm is that it minimises remote access and so minimises the communication costs as each integrity rule is allocated to a site or minimal number of sites to exclude irrelevant sites from the computation of certain rules. Also, the derived distributed rules are simpler and easier to evaluate than the initial constraints as they are evaluated over minimal fragments of relations and so involve less data access. This paper is organised as follows. In Section 2, the basic definitions, notations and examples which are used in the rest of the paper are set out. In Section 3, we discuss the techniques used for enforcing integrity constraints. The integrity rule generation technique adopted by SICSDD is discussed in Section 4. Conclusions and further research are presented in the final Section 5.

## 2.0 PRELIMINARIES

Our approach has been developed in the context of relational databases, which can be regarded as consisting of two distinct parts, namely: an intentional part and an extensional part. A database is described by a database schema, $D$, which consists of a finite set of relation schemas, $<R_1, R_2, \ldots, R_m>$. A relation schema is denoted by $R(A_1, A_2, \ldots, A_n)$ where $R$ is the name of the relation (predicate) with $n$-arity and $A_i$'s are the attributes of $R$. Let dom($A_i$) be the domain values for attributes $A_i$. Then, an instance of $R$ is a relation R which is a finite subset of Cartesian product dom($A_1$) x…x dom($A_n$). A database instance is a collection of instances for its relation schemas. A relational distributed database schema is described as a quadruple ($D$, $IC$, $FR$, $AS$), where $IC$ is a finite set of integrity constraints, $FR$ is a finite set of fragmentation rules and $AS$ is a finite set of allocation schemas.

Database integrity constraints are expressed in prenex conjunctive normal form with the range restricted property [6]. A conjunct (literal) is an atomic formula of the form $R(u_1, u_2, \ldots, u_k)$, where $R$ is a $k$-ary relation name and each $u_i$ is either a variable or a constant. A positive atomic formula (positive literal) is denoted by $R(u_1, u_2, \ldots, u_k)$ whilst a negative atomic formula (negative literal) is prefixed by $\neg$. An (in)equality is a formula of the form $u_1$ OP $u_2$ (prefixed with $\neg$ for inequality) where both $u_1$ and $u_2$ can be constants or variables and OP $\in \{<, \leq, >, \geq, \neq, =\}$.

A set of fragmentation rules, $FR$, specifies the set of restrictions, $C_i$, that must be satisfied by each fragment $R_i$. These rules introduce a new set of integrity constraints and therefore, have the same notation as $IC$. For simplicity, we will consider horizontal fragmentation only. We assume that the fragmentation of relations satisfies the completeness, the disjointness and the reconstructability properties [7]. An allocation schema locates a fragment, $R_i$, to one or more sites. Throughout this paper the same example *company* database is used, as given in Fig. 1.

*Schema*:
emp(eno, dno, ejob, esal);
dept(dno, dname, mgrno, mgrsal);
proj(eno, dno, pno);
*Integrity Constraints (Global Constraints):*
'A specification of valid salary'
*IC*-1: $(\forall w \forall x \forall y \forall z)(emp(w, x, y, z) \rightarrow (z > 0))$
'Every employee has a unique eno'
*IC*-2: $(\forall w \forall x1 \forall x2 \forall y1 \forall y2 \forall z1 \forall z2)(emp(w, x1, y1, z1) \wedge emp(w, x2, y2, z2) \rightarrow (x1 = x2) \wedge (y1 = y2) \wedge (z1= z2))$
'Every department has a unique *dno*'
*IC*-3: $(\forall w \forall x1 \forall x2 \forall y1 \forall y2 \forall z1 \forall z2)(dept(w, x1, y1, z1) \wedge dept(w, x2, y2, z2) \rightarrow (x1 = x2) \wedge (y1 = y2) \wedge (z1 = z2))$
'The *dno* of every tuple in the *emp* relation exists in the *dept* relation'
*IC*-4: $(\forall t \forall u \forall v \forall w \exists x \exists y \exists z)(emp(t, u, v, w) \rightarrow dept(u, x, y, z))$
'Every manager in *dept* '*D*1' earns > £4000'
*IC*-5: $(\forall w \forall x \forall y \forall z)(dept(w, x, y, z) \wedge (w = \text{'}D1\text{'}) \rightarrow (z > 4000))$
'Every employee must earn ≤ to the manager in the same department'
*IC*-6: $(\forall t \forall u \forall v \forall w \forall x \forall y \forall z)(emp(t, u, v, w) \wedge dept(u, x, y, z) \rightarrow (w \leq z))$
'Any department that is working on a project $P_1$ is also working on project $P_2$'
*IC*-7: $(\forall x \forall y \exists z)(proj(x, y, P_1) \rightarrow proj(z, y, P_2))$
*Fragmentation Rules:*
*FR*-1: $(\forall w \forall x \forall y \forall z)(emp_1(w, x, y, z) \rightarrow (z > 0) \wedge (z \leq 10000))$
*FR*-2: $(\forall w \forall x \forall y \forall z)(emp_2(w, x, y, z) \rightarrow (z > 10000))$
*FR*-3: $(\forall w \forall x \forall y \forall z)(dept_1(w, x, y, z) \rightarrow (w = \text{'}D1\text{'}))$
*FR*-4: $(\forall w \forall x \forall y \forall z)(dept_2(w, x, y, z) \rightarrow (w = \text{'}D2\text{'}))$

Fig. 1: The *Company* Static Integrity Constraints

## 3.0 INTEGRITY ENFORCEMENT TECHNIQUES

A database state *D* is said to be consistent if and only if it satisfies the set of integrity constraints, *IC*, denoted by $D \models IC$. A database state *D* may change into a new state $D_u$ when it is updated either by a single update operation or by a sequence of updates (transaction), *u*. If a constraint is false in the new state, i.e. $D_u$ is inconsistent, the enforcement mechanism can either perform compensatory actions to produce a new consistent state $D'_u$, or restore *D* by undoing *u*. This is shown in Fig. 2, where the set of constraints *IC* partitions the space of possible states into two distinct regions, namely: a *legal region* where all constraints in *IC* are satisfied, and an *illegal region* where one or more constraints in *IC* are violated. The initial state of a database is assumed to be in the legal region and an update *u* which falsifies one of the constraints leads to a state in the illegal region. The dashed line (i) in the figure is the process of undoing the update operation *u*, and the dashed line (ii) is the process of bringing the database from this illegal state to a legal state by performing compensatory actions. An additional requirement is that the final state reached by compensating a faulty update operation be chosen within a subspace of states which are as compliant as possible with the original intention of the user who issued the update.

The process described above is known as *integrity constraint enforcement* and consists of the following steps:

(i) generate the integrity tests, which are queries composed from the integrity constraints and the update operations;
(ii) run these queries against the database; and
(iii) depending on the result of the queries, trigger the appropriate actions to make the database consistent. Steps (i) and (ii) here, which check whether all the integrity constraints of the database are satisfied, are referred to as *integrity checking* and can be considered under two broad headings, namely: *detection methods* and *prevention methods* [9].

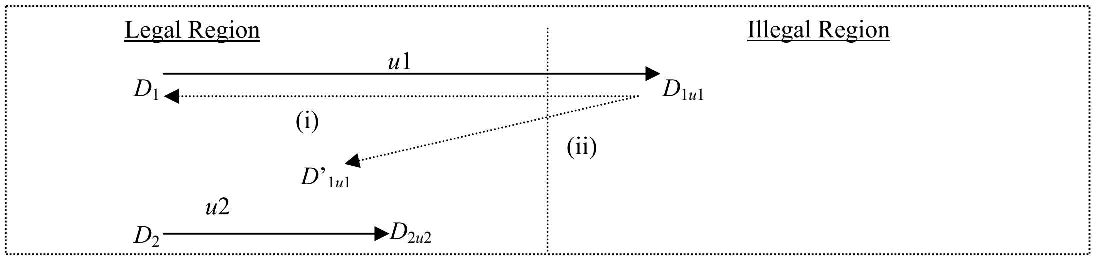

Fig. 2: A Pictorial View of Integrity Constraint Enforcement

The detection methods, which are based upon the concept of *post-tests*, allow an update $u$ to be executed on a database state $D$, which changes it to a new state $D_u$, and when an inconsistent result is detected undo this update. An improvement to the detection method would be to prevent the introduction of inconsistencies in the database. This is achieved by prevention methods, which are based upon the concept of *pre-tests*. These allow an update to be executed only if it changes the database state to a consistent state. Due to the inefficiency resulting from post-testing, the more recent integrity control strategies are based on pre-tests [5, 9].

For a distributed database, integrity constraint checking methods can be classified under two further headings, namely: *global methods* and *local methods*. Global methods which are based upon the concept of global tests, perform constraint checking by accessing data at remote sites, whilst local methods which are based upon the concept of local tests, perform constraint checking by accessing data at the local site. This is shown in Fig. 3[2]. The set of constraints $IC$ partitions the space of possible states into two distinct regions, namely: a *legal region* and an *illegal region*. Within the legal region is a *local legal region* where all constraints in $IC$ are proven to be satisfied by utilising the information stored in this region. Outside this boundary, the set of constraints can either be satisfiable if the database state falls in the legal region or unsatisfiable if the database state falls in the illegal region. With respect to the local legal region, three cases can be considered:

(i) the update operation $u1$ brings a database state $D_1$ to a new consistent state $D_{1u1}$ which is in the same local legal region[3];
(ii) the update operation $u2$ brings a database state $D_2$ to a new consistent state $D_{2u2}$ which cannot be proven to be consistent by the information available in the local legal region; and
(iii) the update operation $u3$ brings a database state $D_3$ to a new state $D_{3u3}$ which is in the illegal region.

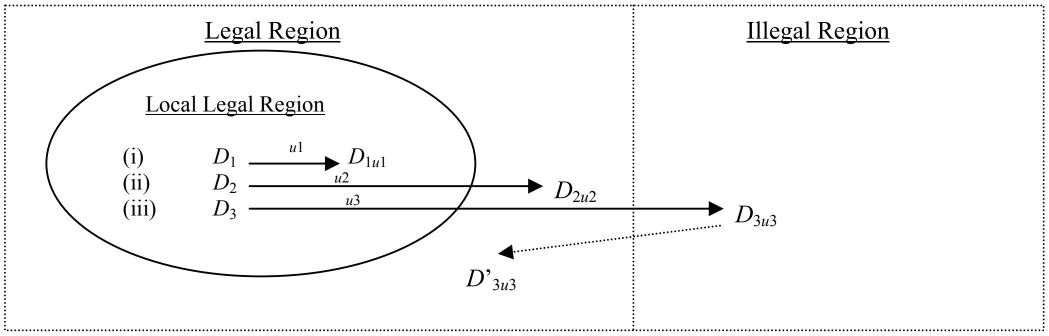

Fig. 3: A Pictorial View of Integrity Constraint Enforcement in Distributed Databases

The integrity tests that are evaluated to verify the consistency of a database within the local legal region are referred to as *local tests*. Since these tests can only identify a subset of legal states (i.e. tests which are sufficient), alternative tests are required, namely those that are evaluated outside the boundary of this local legal region. These tests are referred to as *global tests*. Thus, in a distributed database, four types of integrity tests can be identified. They are global post-tests, local post-tests, global pre-tests and local pre-tests. These tests should possess at least one of the properties mentioned in [5], namely: *sufficient*, *complete* and *necessary*. An integrity test has the sufficiency property if when the test is satisfied, this implies that the associated constraint is satisfied and thus, the update

[2] Figure is based on the observation of *sufficient tests*.
[3] The initial state $D_1$ can also be a legal state outside the local legal region.

operation is safe with respect to the constraint. An integrity test has the necessity property if when the test is not satisfied, this implies that the associated constraint is violated and thus, the update operation is unsafe with respect to the constraint. An integrity test has the completeness property if the test has both the sufficiency and the necessity properties. The classification of integrity tests in distributed databases is summarized in Table 1. For a more precise example, consider the referential integrity constraint and the insert operation given in Table 2. Assume that both relations *emp* and *dept* are allocated to different sites of the network. If the test is not satisfied, then the enforcement mechanism can either undo the insert operation or perform a compensatory action.

Table 1: Classification of Integrity Tests in Distributed Databases

| **Integrity Test Based on Region** | **Integrity Test Based on Detection/Prevention Methods** | **Integrity Test Based on its Properties** |
|---|---|---|
| Global Test<br>- spans remote sites | Post-Test<br>- evaluated after an update is performed | Sufficient Test<br>Necessary Test<br>Complete Test |
| | Pre-Test<br>- evaluated before an update is performed | Sufficient Test<br>Necessary Test<br>Complete Test |
| Local Test<br>- spans local sites | Post-Test<br>- evaluated after an update is performed | Sufficient Test<br>Necessary Test<br>Complete Test |
| | Pre-Test<br>- evaluated before an update is performed | Sufficient Test<br>Necessary Test<br>Complete Test |

Table 2: Examples of Integrity Tests in a Distributed Database

| | |
|---|---|
| *IC-4* | $(\forall t \forall u \forall v \forall w \exists x \exists y \exists z)(emp(t, u, v, w) \rightarrow dept(u, x, y, z))$ |
| Update Operation | insert($emp(a, b, c, d)$) |
| Complete Global Post-Test | $(\exists x \exists y \exists z)(dept(b, x, y, z))$ |
| Sufficient Local Post-Test | $(\exists t \exists v \exists w)(emp(t, b, v, w) \wedge (t \neq a))$ |
| Complete Global Pre-Test | $(\exists x \exists y \exists z)(dept(b, x, y, z))$ |
| Sufficient Local Pre-Test | $(\exists t \exists v \exists w)(emp(t, b, v, w))$ |

The choice between the range of possible tests depends on many criteria such as the application domain, performance requirements, the nature of the database system, etc. Because the input-output volume is generally the most critical factor that influences the performance of the enforcement mechanisms [9] in both centralised databases and distributed databases, most approaches associated with deriving improved integrity tests concentrate on techniques which construct tests that reduce the amount of data accessed during integrity checking. For a distributed database, techniques that construct tests that avoid remote reads and transfer of information across the network also seem attractive. Local pre-tests seem more effective to us than other types of test since:

(i) only a single site is involved in evaluating the local pre-tests;
(ii) as they are evaluated at a target site, this avoids remote reading and the amount of data transferred across the network during integrity enforcement is minimized – in fact, no data transfer across the network is required [3]; and
(iii) they are evaluated before the update is performed, which avoids the need to undo an update in the event of constraint violation and thus, reduces the overhead cost of checking integrity.

## 4.0 INTEGRITY RULE GENERATION TECHNIQUE

The high execution cost of constraint enforcement is one of the major problems in the field of constraint handling. This cost can be substantially reduced not only by applying an efficient enforcement strategy but also by generating a good set of integrity constraints. The techniques incorporated into our system seek to derive *efficient* sets of fragment constraints[4] and a range of possible local tests. Here, 'efficient' means a set of fragment constraints which is semantically equivalent to the initial set, does not contain any semantically or syntactically redundant constructs/constraints, eliminates any constraints which contradict already existing constraints/rules, and contains constraints that are either more local (less distributed) or entirely local when compared to the initial set. Our techniques are identified as *constraint preprocessing* and *constraint distribution techniques*. A full description of these techniques can be found in [4]. Fig. 4 lists the sets of fragment constraints derived after applying the constraint preprocessing techniques to the initial constraints of Fig. 1. Each *FC-i* is a semantically equivalent set of *IC-i*.

Once integrity constraints have been specified for a database and their semantically equivalent sets of fragment constraints have been derived, maintaining a database's integrity whenever the database state changes to a new state involves checking that these constraints are not violated by the operations that caused the transition. Two types of information are required, namely: (i) *when* to enforce constraints, and (ii) *what* to do when a constraint is violated by the database. Thus, a more operational form of an integrity constraint is required. This form is called an *integrity rule*. In our language, an integrity rule has the following template:

WHEN triggering operation
IF NOT integrity test
THEN then-action
[ELSE else-action]

$FC\text{-}1_1$: $(\forall w\forall x\forall y\forall z)(emp_1(w, x, y, z) \rightarrow (z > 0)\ \Lambda\ (z \leq 10000))$
$FC\text{-}1_2$: $(\forall w\forall x\forall y\forall z)(emp_2(w, x, y, z) \rightarrow (z > 10000))$
$FC\text{-}2$: $\Lambda^2_{i=1}\ (\forall w\forall x1\forall x2\forall y1\forall y2\forall z1\forall z2)(emp_i(w, x1, y1, z1)\ \Lambda\ emp_i(w, x2, y2, z2) \rightarrow (x1 = x2)\ \Lambda\ (y1 = y2)\ \Lambda\ (z1 = z2))$
$FC\text{-}3$: $\Lambda^2_{i=1}\ (\forall w\forall x1\forall x2\forall y1\forall y2\forall z1\forall z2)(dept_i(w, x1, y1, z1)\ \Lambda\ dept_i(w, x2, y2, z2) \rightarrow (x1 = x2)\ \Lambda\ (y1 = y2)\ \Lambda\ (z1 = z2))$
$FC\text{-}4$: $\Lambda^2_{i=1}V^2_{j=1}\ (\forall t\forall u\forall v\forall w\exists x\exists y\exists z)(emp_i(t, u, v, w) \rightarrow dept_j(u, x, y, z))$
$FC\text{-}5$: $(\forall w\forall x\forall y\forall z)(dept_1(w, x, y, z) \rightarrow (z > 4000))$
$FC\text{-}6$: $\Lambda^2_{i=1}\ \Lambda^2_{j=1}\ (\forall t\forall u\forall v\forall w\forall x\forall y\forall z)(emp_i(t, u, v, w)\ \Lambda\ dept_j(u, x, y, z) \rightarrow (w \leq z))$
$FC\text{-}7$: $(\forall x\forall y\exists z)(proj_1(x, y, P_1) \rightarrow proj_2(z, y, P_2))$

Fig. 4: The Sets of Fragment Constraints derived by the Constraint Preprocessing Techniques

A rule is triggered when its triggering operation is verified by some database modification. Once a rule is triggered, the integrity test is checked. This is the test generated by our simplification methods. An integrity rule is a *global rule* (*local rule*, respectively) if the test specified in the rule is a global test (local test, respectively). If a test is not satisfied, an action is executed. The action of a local rule consists of two parts, namely: the THEN part, which is performed when the test is not satisfied; and the ELSE part, which is performed when the test is satisfied. The technique employed in our work to derive the integrity rules is called the *integrity rule generation technique* which consists of the three steps described in the following subsections.

### 4.1 Constructing Update Templates

The first step of the integrity rule generation process is the construction of the update templates, *U* which is performed by the update_analysis_procedure. By analyzing each fragment constraint, syntactically, the update_analysis_procedure derives all possible update operations, *U*, that might violate the constraint. Given a constraint specified in prenex conjunctive normal form, the update theorems specify the update operations that will never violate the constraint. The proofs of these theorems can be found in [5, 6] and are therefore omitted here.

[4] A fragment constraint is a constraint which is specified over fragments of relations.

These theorems are as follows:

**Theorem 1**: Whenever an update operation is dealing with the extension of a relation R, integrity constraints in which *R* does not occur are unaffected. In other words, an update operation on a relation R will not violate constraints in which *R* has no occurrences.

**Theorem 2**: Integrity constraints which do not contain *R* in a negated atomic formula are unaffected when a tuple is inserted into the extension of R. In other words, an insert operation on a relation R will not violate constraints in which *R* has no negative occurrences.

**Theorem 3**: Integrity constraints which do not contain *R* in a non-negated atomic formula are unaffected when a tuple is deleted from the extension of R. In other words, a delete operation on a relation R will not violate constraints in which *R* has no positive occurrences.

The derivation of a set of update templates from an integrity constraint specified in prenex conjunctive normal form is performed by a simple syntactical analysis of the constraint. From the update theorems above, the following can be concluded:

- For each negative occurrence of a relation R with *n*-arity in *IC*, an insert template, insert($R(t_1, t_2, \ldots, t_n)$) is generated where $t_i$ is a generic constant corresponding to the attribute $A_i$ of the relation R (from Theorem 2).
- For each positive occurrence of a relation R with *n*-arity in *IC*, a delete template, delete($R(t_1, t_2, \ldots, t_n)$) is generated where $t_i$ is a generic constant corresponding to the attribute $A_i$ of the relation R (from Theorem 3).

Fig. 5 illustrates the update_analysis_procedure. The procedure produces a set of update templates for a given constraint. It recursively traverses the literals of a constraint, and adds the appropriate update template to the resulting update template set whenever it encounters a literal which represents a relation. The resulting update templates set does not contain duplicate operations.

```
Given an integrity constraint IC-i in prenex conjunctive normal form
LET Update_Set$_{IC-i}$ = { }
FOR each occurrence of a relation R with n-arity in IC-i DO
BEGIN
    IF $\neg R$ (the negative occurrence of the relation R in IC-i)
    THEN Update_Set$_{IC-i}$ = Update_Set$_{IC-i}$ $\cup$ insert($R(t_1, t_2, \ldots, t_n)$)
    IF R (the positive occurrence of the relation R in IC-i)
    THEN Update_Set$_{IC-i}$ = Update_Set$_{IC-i}$ $\cup$ delete($R(t_1, t_2, \ldots, t_n)$)
END
```

Fig. 5: The Update_analysis_procedure

Example: Consider the sets of fragment constraints, *FC*-4, of Fig. 4. The possible update templates generated by the update_analysis_procedure are: $(\cup^2_{i=1}\text{insert}(emp_i(a, b, c, d))) \cup (\cup^2_{i=1}\text{delete}(dept_i(a, b, c, d)))$ where *a*, *b*, *c* and *d* are generic constants.

### 4.2 Generating the Integrity Tests

The second step in the integrity rule generation process is to derive the integrity test, *T*, or the simplified forms of the integrity constraints which is performed by the integrity_test_generation_procedure. The procedure consists of two algorithms. The algorithm employed to generate local pre-tests is referred to as Algorithm-B. It is a modification of the algorithm proposed in [6], which is referred to as Algorithm-A. The difference between them is that the tests produced by our algorithm is local pre-tests, while the tests produced by [6] are either global or local post-tests. Both algorithms are based on syntactic criteria and use the substitution, subsumption and absorption rules to generate integrity tests. The derivation of sets of integrity tests from a given integrity constraint specified in prenex normal form and its associated set of update templates is performed by employing both Algorithm-A and Algorithm-B. A full description of these algorithms can be found in [4]. Fig. 6 illustrates the integrity_test_generation_procedure. For a given integrity constraint and its associated set of update operations (i.e. the update templates which are generated automatically by the update_analysis_procedure as presented in Section 4.1), the integrity_test_generation_procedure produces sets of integrity tests which can be global and local tests. This procedure takes an integrity constraint, *IC-i*, and its respective update operation, $U_j$, and generates a set of global and/or local post-tests (shown by Test_Set$_{Uj}$ (Algorithm-A) in Fig. 6) by applying the Algorithm-A and a set of local pre-tests (shown by Test_Set$_{Uj}$ (Algorithm-B) in Fig. 6) by applying the Algorithm-B. The resulting set of tests does not contain any duplicate tests. This process is repeated for each of the update operations associated to *IC-i*.

```
Given an integrity constraint $IC\text{-}i$ in prenex conjunctive normal form
LET Update_Set$_{IC\text{-}i}$ = $\{U_1, U_2, \ldots, U_n\}$
FOR each $U_j$ in Update_Set$_{IC\text{-}i}$ DO
BEGIN
    LET Test_Set$_{Uj}$ = {}
    APPLY Algorithm-A to $IC\text{-}i$ and $U_j$ to generate Test_Set$_{Uj}$ (Algorithm-A)
    APPLY Algorithm-B to $IC\text{-}i$ and $U_j$ to generate Test_Set$_{Uj}$ (Algorithm-B)
    Test_Set$_{Uj}$ = Test_Set$_{Uj}$ $\cup$ Test_Set$_{Uj}$ (Algorithm-A)
    Test_Set$_{Uj}$ = Test_Set$_{Uj}$ $\cup$ Test_Set$_{Uj}$ (Algorithm-B)
END
```

Fig. 6: The Integrity_test_generation_procedure

Example: Consider the sets of fragment constraints, *FC*-4, of Fig. 4 and the possible update templates generated above. The integrity tests generated by the integrity_test_generation_procedure are in Table 3.

Table 3: Generation of integrity tests

| Update | Algorithm-A | Algorithm-B |
|---|---|---|
| $V^2_{i=1}$insert($emp_i(a, b, c, d)$) | $V^2_{j=1}(\exists x \exists y \exists z)(dept_j(b, x, y, z))$ *FC*-4 is violated if the *dno b* does not exist in any of the fragments $dept_j$. | $(\exists t \exists v \exists w)(emp_i(t, b, v, w))$ The existence of *dno b* can be derived from $emp_i$ if there exists at least one employee who is currently working in that department. |
| $V^2_{i=1}$delete($dept_i(a, b, c, d)$) | $\Lambda^2_{j=1}(\forall t \forall v \forall w)(\neg emp_j(t, a, v, w))$ *FC*-4 is violated if there exists at least one employee who is working in *dno a*. | $(\exists x \exists y \exists z)(dept_i(a, x, y, z) \wedge (y \neq \text{null}) \wedge (y \neq c))$ If a null value is not permitted in the system, and if the *dno a* exists in the $dept_i$, this implies that there is a manager who is currently working in $dept_i$. |

## 4.3 Deriving the Violation Actions

The third step in the integrity rule generation process is to derive the appropriate violation action, *A*, for a given integrity constraint and its associated update operation which is performed by the violation_action_procedure. In deriving the violation actions, the constraint designer is involved in choosing an action, *Ai*, to be taken when the test, *Ti*, does not hold. The constraint designer is required to specify an action only when a global rule or a local rule whose test is derived by Algorithm-A is involved. The action for this rule can be one of the following:

- Reject the requested update operation – by means of an ABORT statement.
- Initiate corrective/compensating action – by means of another sequence of data manipulation operations.

For a local rule derived by Algorithm-B, the action, *A*, to be taken is system generated which can be one of the following:

- Reject the requested update operation – by means of an ABORT statement.
- Invoke a global integrity rule – by specifying the integrity rule name.

In practice, the action in a global rule or local rule (Algorithm-A) is specified as ABORT and the action in a local rule (Algorithm-B) is specified with the appropriate global rule name. The selection of these actions is sufficient to produce a consistent database state. The integrity rules generated by the integrity rule generation technique for the sets of fragment constraints given in Fig. 4 are as shown in Table 4 where *GR-FC-i* and *LR-FC-i* are a global rule and a local rule, respectively. As specified in the rules, the action of the global rule *GR-FC-i* in case of constraint violation is either to abort the update operation or to initiate corrective/compensating action while the action of the local rule *LR-FC-i* when the local test is not satisfied is to invoke the global rule *GR-FC-i*.

Table 4: The Integrity Rules derived by the Integrity Rule Generation Technique with respect to the Fragment Constraints given in Fig. 4

| **FC** | **Global Integrity Rule, *GR-FC-i*** | **Local Integrity Rule, *LR-FC-i*** |
|---|---|---|
| $FC\text{-}1_1$ | - | WHEN insert($emp_1(a, b, c, d)$)<br>IF NOT $(d > 0) \wedge (d \leq 10000)$ THEN abort; |
| $FC\text{-}1_2$ | - | WHEN insert($emp_2(a, b, c, d)$)<br>IF NOT $(d > 10000)$ THEN abort; |
| $FC$-2 | WHEN $\bigvee^{2}_{i=1}$insert($emp_i(a, b, c, d)$)<br>IF NOT $\bigwedge^{2}_{j=1}(\forall x \forall y \forall z)(\neg emp_j(a, x, y, z) \vee (x \neq b) \vee (y \neq c) \vee (z \neq d))$ THEN abort; or<br>WHEN $\bigvee^{2}_{i=1}$insert($emp_i(a, b, c, d)$)<br>IF NOT $\bigwedge^{2}_{j=1}(\forall x \forall y \forall z)(\neg emp_j(a, x, y, z) \vee (x \neq b) \vee (y \neq c) \vee (z \neq d))$<br>THEN $\bigwedge^{2}_{j=1}$delete($(\forall x \forall y \forall z)(emp_j(a, x, y, z) \wedge (x \neq b) \wedge (y \neq c) \wedge (z \neq d)$); | WHEN $\bigvee^{2}_{i=1}$insert($emp_i(a, b, c, d)$)<br>IF NOT $(\forall x \forall y \forall z)(\neg emp_i(a, x, y, z))$<br>THEN abort ELSE *GR-FC*-2; |
| $FC$-3 | - | WHEN $\bigvee^{2}_{i=1}$insert($dept_i(a, b, c, d)$)<br>IF NOT $(\forall x \forall y \forall z)(\neg dept_i(a, x, y, z))$ THEN abort;<br>or<br>WHEN $\bigvee^{2}_{i=1}$insert($dept_i(a, b, c, d)$)<br>IF NOT $(\forall x \forall y \forall z)(\neg dept_i(a, x, y, z) \vee (x \neq b) \vee (y \neq c) \vee (z \neq d))$<br>THEN delete($(\forall x \forall y \forall z)(dept_i(a, x, y, z) \wedge (x \neq b) \wedge (y \neq c) \wedge (z \neq d)$); |
| $FC$-4(a)<br><br><br><br><br><br>$FC$-4(b) | WHEN $\bigvee^{2}_{i=1}$insert($emp_i(a, b, c, d)$)<br>IF NOT $\bigvee^{2}_{j=1}(\exists x \exists y \exists z)(dept_j(b, x, y, z))$<br>THEN abort; or<br>WHEN $\bigvee^{2}_{i=1}$insert($emp_i(a, b, c, d)$)<br>IF NOT $\bigvee^{2}_{j=1}(\exists x \exists y \exists z)(dept_j(b, x, y, z))$<br>THEN $\bigvee^{2}_{j=1}$insert($dept_j(b$, null, null, null$)$);<br>WHEN $\bigvee^{2}_{i=1}$delete($dept_i(a, b, c, d)$)<br>IF NOT $\bigwedge^{2}_{j=1}(\forall t \forall v \forall w)(\neg emp_j(t, a, v, w))$<br>THEN abort; or<br>WHEN $\bigvee^{2}_{i=1}$delete($dept_i(a, b, c, d)$)<br>IF NOT $\bigwedge^{2}_{j=1}(\forall t \forall v \forall w)(\neg emp_j(t, a, v, w))$<br>THEN $\bigwedge^{2}_{j=1}$delete($(\forall t \forall v \forall w)(emp_j(t, a, v, w)$); | WHEN $\bigvee^{2}_{i=1}$insert($emp_i(a, b, c, d)$)<br>IF NOT $(\exists t \exists v \exists w)(emp_i(t, b, v, w))$<br>THEN *GR-FC*-4(a);<br><br><br><br><br>WHEN $\bigvee^{2}_{i=1}$delete($dept_i(a, b, c, d)$)<br>IF NOT $(\exists x \exists y \exists z)(dept_i(a, x, y, z) \wedge (y \neq \text{null}) \wedge (y \neq c))$<br>THEN *GR-FC*-4(b); |
| $FC$-5 | - | WHEN insert($dept_1(a, b, c, d)$)<br>IF NOT $d > 4000$ THEN abort; |
| $FC$-6 | WHEN $\bigvee^{2}_{i=1}$insert($emp_i(a, b, c, d)$)<br>IF NOT $\bigwedge^{2}_{j=1}(\forall x \forall y \forall z)(\neg dept_j(b, x, y, z) \vee (d \leq z))$<br>THEN abort; | WHEN $\bigvee^{2}_{i=1}$insert($emp_i(a, b, c, d)$)<br>IF NOT $(\exists t \exists v \exists w)(emp_i(t, b, v, w) \wedge (w \leq d))$<br>THEN *GR-FC*-6; |
| $FC$-7(a)<br><br>$FC$-7(b) | WHEN insert($proj_1(a, b, P_1)$)<br>IF NOT $(\exists z)(proj_2(z, b, P_2))$ THEN abort;<br>WHEN delete($proj_2(a, b, P_2)$)<br>IF NOT $(\forall x)(\neg proj_1(x, b, P_1))$ THEN abort; or<br>WHEN $\bigvee^{2}_{i=1}$delete($proj_2(a, b, P_2)$)<br>IF NOT $(\forall x)(\neg proj_1(x, b, P_1))$<br>THEN delete($(\forall x)(proj_1(x, b, P_1)$); | WHEN insert($proj_1(a, b, P_1)$)<br>IF NOT $(\exists z)(proj_1(z, b, P_1))$ THEN *GR-FC-7*(a);<br>WHEN delete($proj_2(a, b, P_2)$)<br>IF NOT $(\exists z)(proj_2(z, b, P_2) \wedge (z \neq a))$<br>THEN *GR-FC*-7(b); |

Note: a, b, c and d are generic constants.

To detect the existence of cyclic compensating actions, a triggering graph is constructed. The nodes of the graph correspond to the integrity rules in the set. If the execution of an integrity rule, *IR-i*'s action can trigger integrity rule *IR-j* ($i \neq j$) then a directed edge from node *IR-i* to node *IR-j* is constructed. An infinite action is identified when a cycle in the graph is detected, i.e. a cycle is a path through the triggering graph in which a given integrity rules appears more than once. This is corrected by modifying the actions in the relevant rules appropriately. Example of cyclic actions is as follows:

*IR*-1: WHEN *u*1 IF NOT *Test_u*1 THEN *u*2;
*IR*-2: WHEN *u*2 IF NOT *Test_u*2 THEN *u*1;

Fig. 7 illustrates the violation_action_procedure which verifies if a cyclic compensating action has occurred. To check if an action $A_i$ of an integrity rule $IR_i$ produces a cyclic path in the triggering graph, the violation_action_procedure recursively accumulates the integrity rules which are either directly or indirectly invoke by $IR_i$, and a cyclic path is encountered when $IR_i$ is being invoke again.

```
Given an integrity constraint IC-i and an update $U_i$
LET selected_test = $Test_i$ % Test is selected by a constraint designer
IF selected_test = GT or LT_A % A global test or a local test derived by Algorithm-A
THEN get the compensating action, $A_i$ % Specify by the constraint designer
ELSE
IF selected_test = LT-B % A local test derived by Algorithm-B
THEN generate the THEN and ELSE part
    GET compensating action, $A_i$, for GT and LT_A % Specify by the constraint designer
Check for existence of cyclic action
IF NO cyclic action
THEN IRgr = ($Test_i$, $U_i$, $A_i$) % Global rule/Local rule (Algorithm-A)
     IRlr = ($Test_i$, $U_i$, THEN, ELSE) % Local rule (Algorithm-B)
ELSE invalid rule

* Check for the existence of cyclic action
Given an integrity rule with an update operation $U_i$ and an action $A_i$
RETRIEVE all IR's whose update = $A_i$ and LET it be IR_set
REPEAT
      RETRIEVE all actions in IR_set and LET it be IR_action_set (which does not include ABORT operation)
      IF $U_i \in$ IR_action_set
      THEN cyclic = FOUND
      ELSE
      BEGIN
          Take and remove an action, $A_j$, from IR_action_set
          RETRIEVE all IR's whose update = $A_j$ and LET it be IR_set
      END
UNTIL cyclic = FOUND OR IR_action_set = {}
```

Fig. 7: The Violation_action_procedure

When a user requests an update, only those rules that might violate the update are selected for evaluation. Heuristics are employed to reschedule the execution of the integrity rules, thereby enabling early update abortion in the case of constraint violation. The heuristics applied are:

(i) Choose a local integrity rule, i.e. an integrity rule that can be evaluated at a local site.
(ii) Choose an integrity rule whose violation implies that no other integrity rules are triggered. For example, an integrity rule for a key constraint with an ABORT action.
(iii) An integrity rule with test *Ti* which subsumes another integrity rule with test *Tj* is preferred since the truth of the test *Ti* implies the truth of test *Tj* (but not vice versa). For example, if the test of *LR-FC*-6 is satisfied then so is the test *LR-FC*-4(a). In general, the following algorithm for processing the integrity rules is applied when an update request is made:

```
FOR each update request, U,
Let L be the set of all local rules
      whose triggering operations are U
Choose a local integrity rule, LR-i, from set L
REPEAT UNTIL no triggering rules remain:
   Evaluate LR-i's test
```

```
IF test result is false,
    execute LR-i's action
```

## 5.0 SUMMARY

In this paper, we have described an enforcement algorithm based on the rule mechanism for a distributed database which aims at minimising remote access by fully exploiting the available information at a target site. This strategy is valuable in a distributed database where the cost of accessing remote data for verifying database consistency is the most critical factor influencing the performance of the system.

There are a number of extensions and improvements that could be made, as follows:

(i) the types of constraints considered are the types that are widely used in theory and practice, namely: domain, key, referential and simple general constraints – a broader range of constraint types such as aggregate constraints and transition constraints can be considered; and

(ii) further investigation of the effect of the fragmentation and allocation strategy on the derived integrity rules would be worthwhile.

## REFERENCES

[1] S. Ceri, P. Fraternali, S. Paraboschi and T. Tanca, "Automatic Generation of Production Rules for Integrity Maintenance". *ACM Transactions on Database Systems*, Vol. 19, No. 3, 1994, pp. 367-422.

[2] P. W. P. J. Grefen, "Design Considerations for Integrity Constraint Handling in PRISMA/DB1". *Prisma Project Document P508*, University Twente (Netherlands), 1990.

[3] A. Gupta and J. Widom, "Local Verification of Global Integrity Constraints in Distributed Databases", in *Proceedings of the 1993 ACM SIGMOD Conference*, Washington DC (USA), May 1993, pp. 49-58.

[4] H. Ibrahim, W. A. Gray, and N. J. Fiddian, "SICSDD: Techniques and Implementation", in *Proceedings of Constraint Databases and Applications, Second International Workshop on Constraint Database Systems (CDB'97)*, Delphi (Greece), 1997, pp. 187-207.

[5] W. W. McCune. and L. J. Henschen, "Maintaining State Constraints in Relational Databases: A Proof Theoretic Basis". *Journal of the Association for Computing Machinery*, Vol. 36, No. 1, January 1989, pp. 46-68.

[6] J. M. Nicolas, "Logic for Improving Integrity Checking in Relational Data Bases". *Acta Informatica*, Vol. 18, No. 3, July 1982, pp. 227-253.

[7] M. T. Ozsu, and P. Valduriez, "Principles of Distributed Database Systems". *Prentice-Hall International Editions*, 1991.

[8] X. Qian, "Distribution Design of Integrity Constraints", in *Proceedings of the 2nd International Conference on Expert Database Systems*, California (USA), 1989, pp. 205-226.

[9] E. Simon, and P. Valduriez, "Integrity Control in Distributed Database Systems", in *Proceedings of the 19th Hawaii International Conference on System Sciences*, Hawaii (USA), January 1986, pp. 622-632.

## BIOGRAPHY

**Hamidah Ibrahim** obtained her PhD from the University of Wales, Cardiff in 1998. Currently, she is a lecturer at the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. Her research areas include distributed databases and knowledge based system. She has published a number of papers related to these areas.