On the Modes and Meaning of Feedback to Transaction Designers

David Stemple Subhasish Mazumdar
Tim Sheard
University of Massachusetts, Amherst*

ABSTRACT

An analysis of database transactions in the presence of database integrity constraints can lead to several modes of feedback to transaction designers The different kinds of feedback include tests and updates that could be added to the transaction to make it obey the integrity constraints, as well as predicates representing post-conditions guaranteed by a transaction's execution We discuss the various modes, meanings, and uses of feedback We also discuss methods of generating feedback from integrity constraints, transaction details and theorems constituting both generic knowledge of database systems and specific knowledge about a particular database Our methods are based on a running system that generates tailored theories about database systems from their schemas and uses these theories to prove that transactions obey integrity constraints

*This paper is based on work supported by the National Science Foundation under grant DCR-8503613

1. Introduction

> *"It is one thing to shew a Man that he is in Error, and another, to put him in possession of Truth"* — John Locke, 1690

Database transactions are programs, generally short, which update databases in an atomic manner, i e , they either execute completely or they have no visible effect on the database They are also required to respect certain invariants on the database, the database integrity constraints These constraints are expressed as part of a global type definition, the database schema, which declares all the structure of the database Schemas are often split into levels a global, high level structure, a so-called external level which describes different applications' views of the database, and an internal level describing the physical data structures used to store the database The integrity constraints that interest us are those expressed in the high level, global declaration, which is what we will mean whenever we use the term schema in this paper Integrity constraints in the schema express conditions which define *legal* or *consistent* instances of the database and include such conditions as relation keys, subset dependencies, and redundancies such as those caused by maintaining sums and averages of stored data

Proving that a given transaction obeys all the database integrity constraints is a problem in program verification on which we have reported elsewhere [Sheard and Stemple 86] Whenever it cannot be proven that a transaction obeys all constraints, tests can be added to the transaction to transform it into a *safe* transaction, i e , one which obeys all

constraints However, the transformed transaction may not be the transaction that the designer had in mind In fact, the transformed transaction may never be able to execute at all if the tests are guaranteed to fail on all consistent states of the database Often, incorrect transactions can be corrected by the addition of updates rather than of tests However, adding either tests or updates can fail to make the transaction conform to the designer's purpose The designer may simply have made an error, of commission or omission, that obscures his or her intent In this case, the role of analysis should be to give the designer feedback which makes the implications of the mistake clear enough to illuminate the error The modes of such feedback are the subject of this paper We discuss the different modes possible and how they can be used by designers in producing safe transactions which perform the intended updates

Integrity constraints provide a rich semantic environment in which transactions can be mechanically analysed We have developed a theorem prover which analyses transaction programs with the goal of proving that they are safe When the analysis fails, unprovable subgoals are left in a form which allows the addition of tests to the transactions The prover works with a theory comprising theorems which function basically as rewrite rules These rewrite rules and others like them are the raw materials from which feedback to transaction designers can be produced In this paper, as we discuss the various modes of feedback, we also give examples of the kinds of theorems needed for the different forms of feedback

The prover we use is our version of the Boyer-Moore theorem prover[Boyer and Moore 79] extended to higher order functions and theorems The theory that our prover works with is based on a small set of axioms defining four base abstract data types tuples, lists, finite sets and natural numbers This theory constitutes a general theory of database systems, a part of which covers relational database theory It differs from most other database theory in concentrating on theorems about the interaction between complex updates and integrity constraints, rather than on algebraic properties of schemas It is rooted in recursive functions in the manner of, and inspired by, Boyer and Moore but is extended to accommodate strong typing and polymorphism of types While we use the inductive theorem prover for mechanically proving and building our theory, the transaction safety prover uses only rewriting techniques Details of this theory and its use in proving safety theorems appear elsewhere [Stemple and Sheard 83], [Stemple and Sheard 85], [Sheard and Stemple 85], [Stemple et al 86], and [Sheard and Stemple 86] In this paper, we will present enough of the theory and analysis techniques to explain how we mechanically produce the different kinds of feedback

The paper is organized as follows We first present the class of integrity constraints and transactions that we currently handle in our safety prover We then discuss integrity check generation and the way we use theorems in this endeavor Following this we present the main other method of fixing transactions – adding updates or *update propagation* Next we investigate the problem of finding inconsistencies between a transaction and constraints We then discuss the most difficult kind of feedback – generating transaction postconditions that tell the designer in useful terms what the transaction does

2. Integrity Check Generation

In this section we concentrate on the process of generating checks that can be added to a faulty transaction to ensure that it obeys the integrity constraints on its database These checks are to be executed at run-time and can be preconditions, postconditions or executed in the middle of transactions Their failure must cause their transaction to abort We first introduce the transaction and constraint classes we will use in the remainder of the paper Then we discuss the concept of *constraint protectors* and follow this with an analysis of the problem of optimizing constraint protectors by the use of certain kinds of theorems or lemmas

2.1 Constraint classes

We have built a database theory which encompasses a set of database type construction operations, a set of con-

straint mechanisms, and a set of transaction structuring primitives In the current system the following type constructors are allowed

Classification: type-name2 = type-name1
Derivation: type-name2 = type-name1
where derivation-predicate
Tupling: tuple-type-name =
[component1-name component1-type-name,
component2-name component2-type-name,
componentn-name componentn-type-name]
Grouping: type-name2 = *set of* type-name1

These can be composed in arbitrary ways to build a database type A typical first normal form relational database is described by classifying domains using some primitive types, using tupling to define tuple types, and using the tuple types as the basis for group types comprising the relations of the database Finally the database type is formed by tupling the relation types At any stage in forming the database type, derivation can be used to introduce integrity constraints on the component whose type is being defined For example, domain constraints are specified in where clauses of the domain type declarations Key constraints occur in relation type definitions, and interrelational constraints, such as referential integrity, are specified in the where clause of the database type declaration

The constraint classes which may be expressed in where clauses include

- arithmetic constraints
- set membership
- set containment
- null intersection of two sets
- universal quantification
- existential quantification
- redundancy of information stored in a tuple with a function evaluated on a set, an aggregate constraint is an example of this

We now introduce a particular database that will be a source of examples in this paper The database describes the workings of a job matching agency Entities in the database include people, who apply for jobs and are placed with companies Companies offer positions and hire people An entity relationship (ER) diagram extended with a few constraints for the database is shown in Figure 1 and the schema is detailed in Figure 2 We implement this ER diagram as a relational schema by implementing each entity and relationship as a relation The ER diagram imposes several integrity constraints on the relational implementation which are expressed in the where clause of the database type declaration The ovals in the ER diagram indicate relational and inter-relational constraints which we also want the system to enforce These constraints are as follows [1]

1 A person should never simultaneously be placed in a job and have an application for a job, i e only unemployed persons can apply for jobs

2 Each company shall offer jobs for which they have one or more openings A company should never offer a job with zero openings

3 The *placed* field of the *persons* relation is a redundant field That is to say it could be computed by testing if the person is in the *placements* relation We will store this value redundantly for quicker access, and we want the system to ensure that the two representations of this fact always agree

4 In a similar manner the *totsal* component of each company tuple in the *companies* relation should agree with the sum of all the company's employees' salaries in the placements relation

2.2 Transaction classes

Currently our verification system deals with transactions built from the following constructs

- *header* statement with type declarations for input
- *preconditions* expressed in the constraint language

[1] The numbered boxes in Figure 1 refer to this list

- *if-then-else* statements
- *insert*, *delete* and *modify* statements
- four *looping* constructs
  1. delete from a set all elements that obey a specified condition
  2. modify those elements from a set which obey a specified condition
  3. insert functions of the elements of one set into a second set
  4. change a specified subset of a set by modifying those elements meeting some condition and deleting those that do not

Figure 3 gives a transaction (hire) which uses nearly all these constructs This transaction involves placing an applicant in a job which is currently offered For this transaction to be safe it must update five separate relations in the correct manner to ensure integrity In addition to adding the person to the *placements* relation, his/her *placed* field must be set to true in the persons relation In the offerings relation the number of positions open must be decremented, or the tuple removed if the number falls to zero The application for the hiree must be removed, and the totsal of the hiring company must be adjusted

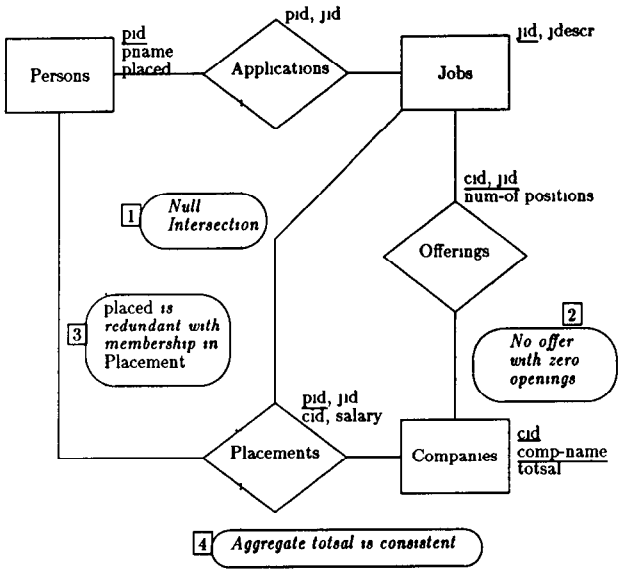

Figure 1: Extended E-R Diagram for Job Agency Database

```
type
  person= [ pid  number, pname  string, placed  boolean ],
  person-rel= set of person where key(person-rel, pid),
  job=[ jid  number, jdescript  string ],
  job-rel= set of job where key(job-rel, jid),
  company=
    [ cid  number, comp-name  string, totsal  number],
  company-rel= set of company
    where key(company-rel, cid),
  application= [pid  number, jid  number],
  application-rel= set of application,
  offer=
    [ cid  number, jid  number, num-of-posns  number ]
    where num-of-posns ≠ 0 ,
  offer-rel= set of offer where key(offer-rel, cid, jid),
  placement= [ pid  number, jid  number,
    cid  number, salary  number ],
  placement-rel= set of placement
    where key(placement-rel, pid),
  database job-agency
    [ persons  person-rel,
      jobs  job-rel,
      applications  application-rel,
      offerings  offer-rel,
      companies  company-rel,
      placements  placement-rel ]
    where
      contains(persons pid, placements pid) ,
      contains(companies cid, placements cid) ,
      contains(jobs jid, placements jid) ,
      contains(jobs jid, offerings jid) ,
      contains(companies cid, offerings cid) ,
      contains(persons pid, applications pid) ,
      contains(jobs jid, applications jid) ,
      for x in persons
          x placed is-redundant-with
          x pid in placements pid ,
      null-intersection (applications pid, placements pid),
      for c in companies
          c totsal is-redundant-with
          sum( (all p in placements where p cid = c cid),
            salary),
```

Figure 2: Schema of job-agency

```
transaction hire (comp, hiree, jb, sal  number ) ,
preconditions
        hiree in persons pid ,
        [comp, jb] in offerings [cid, jid] ,
        hiree not-in placements pid ,
begin

update p in persons where
  p pid = hiree by [ placed  = true ] ,
update o in offerings where
  o cid = comp and o jid = jb by
    if o number-of-positions = 1
    then delete o
    else [ number-of-positions  = number-of-positions - 1 ] ,
insert [hiree, jb, comp, sal] into placements,
remove p from applications where p pid = hiree,
update c in companies where c cid = comp
  by [ totsal  = totsal + sal ]
end,
```

Figure 3: Hire Transaction

2.3 Lemmas, higher-order functions, and meta-lemmas

Our transaction safety verification system consists of several theorem provers built on top of the kernel of a Boyer-Moore theorem prover Much of the reasoning done in these provers is performed by heuristically applying theorems (which we nearly always call lemmas) which contain *rewrite rules* A lemma is of the form

$$(L = R) \Leftarrow H,$$

where H is a hypothesis (conjunction of zero or more predicates) and (L = R) is a rewrite rule stored as an equation and used in a left to right manner (the left-hand side L is to be replaced by the right-hand side R if the hypothesis H can be proved to be true with the current set of truths) [2] Note that a function definition is a kind of rewrite rule for rewriting calls of the function in terms of the function body The processing of transactions to provide feedback also makes extensive use of rewrite lemmas

Higher order functions are used to encapsulate structured updates and integrity constraint primitives We have also used higher order functions and associated lemmas to capture knowledge common to groups of predicates sharing fundamental structure such as relational selection, set containment and universal quantification Lemmas about such groups of functions are called meta-lemmas and are characterised by function-variables in their bodies

We have defined key, update, null-intersection, remove, and a number of other important functions, both updating and predicate functions, as recursive functions over sets with functional parameters For example, the update function takes as input a set r, a predicate !p which tells which tuples to change, and a tuple transforming function !f (as well as some optional extra inputs &x) We use the convention that a variable that starts with an exclamation mark (!) is a function variable, and that one starting with an ampersand (&) is optional The following defines a generic set update function It is in an internal prefix form of pure LISP that is used by the system in all its reasoning Note that ADABTPL (see Figs 2 and 3) is the user-level language, and its semantics are expressed in a pure LISP form

```
(fun (name update)

(generic-types alpha &extra)

(params (r (set of alpha))
        (!f (function
              ((cross alpha &extra)
                 where (x) (!p ($1 x) ($2 y))
              alpha))
        (!p (function (cross alpha &extra) boolean))
        (&x &extra))

(body   (if (equal r emptyset)
            emptyset
            (if (!p (choose r) &extra)
                (insert (!f (choose r) &extra)
                        (update (rest r) !f !p &extra))
                (insert (choose r)
                        (update (rest r) !f !p &extra)))))

(result (set of alpha)))
```

The definition can be read as follows The function name is update It knows about two arbitrary generic types, alpha and &extra The function has four parameters The first is a set of alpha, where alpha can be any type since it is generic The second and third parameters are functions

[2] Rules not in this form can always be recast this way For example, p(x) becomes p(x) = true and not q(y) becomes q(y) = false

!p is a predicate from (alpha, &extra) to boolean, and !f is a function from (alpha, &extra) to alpha The where part of the function type declaration for !f states that (!f x &y) may only be called where (!p x &y) is true This is an example of the very strong typing which the system enforces The body of the function is the recursive expression in the body clause The form of the body is prefix pure LISP with all expressions starting with a function name followed by the function parameters For example, f(x,y) is written (f x y) This includes the basic computational logic operator *if*, which is always written (if condition then-expression else-expression) The functions choose and rest are the finite set correlates of car and cdr respectively The result line declares that the type of the result that update returns is a set of alpha

Meta-lemmas have the following form

$$(L = R) \Leftarrow H \quad \text{IF MH}$$

One may read the above as Lemma IF MH, or equivalently, Meta-body IF MH, where MH stands for meta-hypothesis and is a condition on the functional parameters in the generic functions contained in the meta-body, (implies H (equal L R)) A meta-body is to be used eventually as a rewrite lemma Specific lemmas generated from a meta-lemma consist of the meta-body with its functional variables replaced by concrete function names or lambda expressions

For example a meta-lemma about update, *update-preserves-key*, is given below

(key (update s !f !p &m) !c) = true $\Leftarrow$ (key s !c)
IF (!c (!f x &m)) = (!c x)

Its meta-hypothesis describes properties of !f (a functional input to update), and the meta-body describes a property of update if the meta-hypothesis holds The meta-lemma states that if !f does not change the column !c in s on which a key constraint is declared, then the update does not change the property of s being keyed on !c

2.4 Constraint protectors

In order to prove that a transaction is safe, we assume that the database is consistent when the transaction begins and try to prove that each constraint is true when the transaction has finished If we fail to prove that a constraint has to be true at transaction end, a test can be added to test that constraint at run-time The test could be the constraint itself, but in general can be simpler and faster We call such a test a ***constraint protector*** The generation of constraint protectors has been studied by many researchers [Bernstein and Blaustein 81], [Bernstein and Blaustein 82], [Nicolas 82], [Henschen et al 84], [Hsu and Imielinski 85], [Bunker 86] Our approach is most closely related to that of Henschen et al who also use a theorem prover to generate constraint protectors The major differences between their work and ours are that their updates are much simpler than ours and they use a resolution-based Horn clause prover to generate sufficient tests, whereas we use a Boyer-Moore style prover to generate necessary and sufficient tests followed by a phase to generate better tests, some of which may be only sufficient

For example, consider the integrity constraint

(contains r1 r2)

and a transaction that deletes an element from r1 We need to prove that the integrity constraint is still valid after the deletion, i e , that the ***updated integrity expression***

(contains (delete a r1) r2)

holds given that the integrity constraint held on the original database Thus, we need to prove the following

(contains (delete a r1) r2) $\Leftarrow$ (contains r1 r2)

To prove the above formula, the system applies the rewrite rule

(contains (delete a r1) r2) = (not (member a r2))
$\Leftarrow$ (contains r1 r2)

and arrives at the unsimplifiable formula

(not (member a r2))

which is also the necessary and sufficient test for the validity of the updated integrity expression given that the integrity constraint held in the original database

2.5 Use of lemmas for optimization

In our verifier, the transaction safety theorem is reduced first to conjunctive normal form and each conjunct is treated as a separate subgoal Each subgoal (an updated integrity expression) is reduced by successively rewriting subterms using *equality* rules (i e , rules of the form *lhs* = *rhs*) chosen heuristically If the term cannot be reduced to true, the residual term obtained represents a necessary and sufficient test for verifying an integrity constraint It may however be cheaper (in terms of run-time testing) to arrive at a *sufficient condition* that assures the same predicate if true Towards this end, the system attempts to *back-chain* from the residue and output a list of sufficient conditions

Consider a variation of the previous example in which the transaction remains as before, but the constraint becomes

(contains (project r1 pid) (project r2 pid))

The test obtained is

(not (member (pid a) (project r2 pid)))

If there is a lemma of the form ((test = true) ⇐ hypoth), then hypoth can be a sufficient test The process of back-chaining continues until there are no more lemmas to unify the test with, basically in the same manner employed by Henschen et al [Henschen et al 84] Indeed the lemmas

1 (not (member x s)) ⇐ (equal s emptyset), and
2 (equal (project s 'c) emptyset) ⇐ (equal s emptyset)

lead to the tests

1 (equal (project r2 pid) emptyset),and
2 (equal r2 emptyset)

Now consider a situation where r2 has a type restriction (e g , the pid of each element must be less than 1000) as part of the integrity constraint

(for-all (project r2 pid) less-than-thousand),

where (less-than-thousand x) = (lessp x 1000)
The process of back-chaining proceeds as follows The test unifies with the conclusion of the following lemma

(not (member x s)) ⇐ (for-all s 'p), (not ('p x)))

(which says that if all elements of a set meet a certain predicate not satisfied by x, then x cannot belong to the set), yielding a binding for x and s Next the hypotheses of the lemma is examined The first literal (for-all s 'p) unifies with the above integrity constraint binding 'p Thus the remaining literal (not ('p x)) once instantiated leads to another sufficient test

(not (lessp (pid a) 1000)

Such sufficent tests are useful in those cases where the condition tested, emptyness of relations in the first examples, and a type check on the deleted data element in the last are inexpensive and occur often enough to reduce the overall cost Obviously, they cannot replace necessary conditions

3. Update Propagation

Adding tests may not be the right thing to do to a transaction Missing tests may simply be a symptom of another problem For example, the transaction writer may have forgotten updates Integrity constraints often require that updates be grouped in order to rationally update a database to model a real world event In this section, we discuss two common cases of this phenomenon, caused by natural hierarchies and storing information representing aggregates of other stored information

In order to suggest updates to be added, we employ a limited kind of functional unification The basic idea is as follows Given an integrity constraint of the form

(predicate r1 r2),

and a delete from r2 (say), if we find that the updated integrity expression

(predicate r1 (delete a r2))

is not provably true, we pose the question,"for what value of the function variable [3] 'x, is the expression

(predicate ('x r1) (delete a r2))

true?" The actual mechanism for finding the solution is

[3]The undecidability of higher-order unification is not an issue here We merely pattern match this variable with our finite lemma base

PROLOG-like and will be illustrated in the following sections

3.1 Deleting down a hierarchy

One common example of update propagation is a hierarchy in which one deletion must give rise to others Consider the previous example in which containment of r2 in r1 is part of a hierarchy The transaction consisting solely of the deletion of an element of r1 as before gives rise to the test

(not (member a r2))

Now we rewrite the above, replacing sets (r2 in the above example) by variables and attempt to find bindings for which it can be proven to be true

The lemma

(not (member x (delete x r)))

unifies with the answer r2 ← (delete a r2), and this says that an additional update to relation r deleting the element a from the smaller set r2 would make the transaction safe

Consider the case where the constraint is referential integrity, specified by our higher-order function *refer* [4]

(refer r1 'c1 r2 'c2)

which means that the projection on the 'c1 attribute of the r1 relation must contain the projection on the 'c2 attribute of the r2 relation Now given a transaction that attempts to delete an element from r1, we need to prove

(refer (delete a r1) 'c1 r2 'c2) ⇐ (refer r1 'c1 r2 'c2)

Using a rewrite rule, the following residual term is obtained

```
(if (member ('c1 a) (project r2 'c2))
    (member ('c1 a) (project (delete a r1) 'c1))
    true)
or, in clausal form
    (not (member ('c1 a) (project r2 'c2)))
    ∨ (member ('c1 a) (project (delete a r1) 'c1))
```

Now the system attempts to backchain from each literal, disregarding those in which the only updated set is r1

The idea behind the heuristic is that since the user has specified this particular update, the one he/she *missed* must affect some other relation Note in the case of real referential integrity as in our job-agency example, 'c1 would be a key on r1 and so the second literal would be false and can be eliminated The lemma that the first literal unifies with is the following

```
(not (member x
             (project (remove s f x)
                      'c)
where (f x y) = (equal ('c x) y)
```

This lemma states that after removal from a set of all tuples whose c-attribute is equal to x its projection on the c-attribute does not contain x Unification binds x to ('c1 a), 'c to 'c2, and the suggested binding for r2 gives the suggested update

```
(remove r2 h ('c1 a))
where (h x y) = (equal ('c2 x) y)
```

In other words, an update that, if added, would cause the constraint to be obeyed is the removal from r2 of tuples whose c2-component is equal to the c1-component of the deleted r1-tuple

3.2 Additional Updates: Use of Meta-lemmas

The mechanism for update propagation becomes more complicated when meta-lemmas are involved The basic idea is that the conclusion of the meta-lemma suggests the update and the meta-hypothesis gives us the constraints on the variables involved in the update

Consider the constraint in the job-agency example describing that the placed field of every person in the persons relation is true if and only if that person is included in the *placements* relation This constraint is translated using the redun function (Recall that the expression (redun r1 'f 'g r2) means ∀x∈r1 'f(x) = 'g(x,r2)) in the following way

(redun persons placed g placements)

where (g x r) = (member (pid x) (project r pid))

[4] This function is more general than needed by referential integrity

This means that

∀ p∈*persons*
p placed ≡ (p pid ∈ project (placements, pid))

Suppose a transaction only changes a certain person's (whose pid equals *name*) *placed* field to true, leaving out the insertion into the placements relation, we begin examining the updated integrity expression

(redun (update persons transf pred) placed g placements),

where
(transf x) = (person (pid x) (pname x) true)[5],
(pred x) = (equal (pid x) name)

Now, examining the meta-lemmas, we find the following

(redun (update r1 !t !p) !f !g (insert newtuple r2)) = true
⇐ (redun r1 !f !g r2)
The meta-hypothesis is given by
∀x∈r1
(if (!p x)
(!g (!t x) (insert newtuple r2)) = (!f (!t x))
(!g x (insert newtuple r2)) = (!g x r2))

The consequent of the meta-lemma says that the redun constraint is maintained under an update to r1 *and* an insertion to r2 provided the meta-hypothesis is valid This meta-hypothesis requires that whenever the update to r1 changes a tuple x to (!t x), the relation between the !g function and the !f function holds, and for those tuples that are unaffected by the update, the !g function is unaffected by the insertion into r2 Since this meta-lemma is a candidate for suggesting additional updates, its meta-hypothesis is examined If a contradiction emerges, then it is rejected, otherwise it yields some (sufficient) constraints on the unknown data elements (newtuple) involved in the additional update (insert) suggested by this meta-lemma The *else* part of the hypothesis does not lead to useful constraints, but the *then* part leads to[6]

(member
name
(insert (pid newtuple) (project placements pid)))

This suggests that safety can be attained through the inclusion of an update that inserts into placements a tuple *newtuple* meeting the above constraints

4. Inconsistencies and Redundant Checks

In tightly constrained databases with redundancy constraints, it is possible for a subgoal to *evaluate to false* This is a very interesting result It means that no addition of tests will make the transaction safe Further, our earlier technique of finding additional updates fails since the subgoal evaluates to false and there are no set variables to unify with lemmas However, redundancy constraints normally mean that one kind of update on one relation needs necessarily to be accompanied by an update on another This is formalised by lemmas about these constraints (the earlier redun-meta-lemma for example) Thus we modify the earlier technique of finding updates as follows

Consider an updated integrity expression which is of the form

(pred (f r1) (g r2))

where f and g are update functions We assume that one of the update functions (say g) is wrong and needs to be changed, and based on that premise, we attempt to evaluate the proper value of g by rewriting the subgoal as

(pred (f r1) r2))

and then attempt to find the necessary update on r2 as outlined earlier If this succeeds, the system reports that the update function g on r2 if *replaced* by a different function (suggested by the meta-lemma) would make the transaction safe If it fails, we attempt to do the same with f

Further, we check to see if more than one such predicate evaluates to false under the given transaction If there are two different predicates pred1 and pred2 leading to the following updated integrity expressions

(pred1 (update r1) (rem b r2)) = false,
(pred2 (update r3) (rem b r2)) = false,

[5]person is the function which constructs a tuple of person tuple

[6]We will report on the constraint simplifier elsewhere

then, since the remove function is the *only* update function *common* to both expressions, we may guess that that is the wrong update function If the user had made just one mistake, that indeed would have to be the one

Such is the situation in our job-agency example if pred1 is null-intersection and pred2 is redun If a transaction changes the placed field of a person in the persons relation to false, inserts a tuple into the applications relation, and also inserts a tuple (instead of deleting) into the placements relation, the error will be caught since both constraints will evaluate to false and the insert into r2 will be the only common update function in the two expressions

Another source of redundant checks, is the transaction program itself The user, owing to oversight, may include checks (typically in *if* statements) that are eliminable Discovering these is a useful exercise since it can cut down on run-time testing By analysing the body of the transaction program, it is possible to make a number of simplifications For example, given a line

$$(\text{if } p_1 \text{ then } (\text{if } p_2 \text{ then } e_1 \text{ else } e_2),$$

an attempt is made to evaluate the conditions p_1 and p_2 using conditions which must hold at the place where this line is to be evaluated Suppose, whenever the integrity constraints hold on the current database state, and the preconditions of the transaction and the condition p_1 are all true, p_2 has to be true Then the above line of code may be replaced by (if p_1 then e_1), thus avoiding the computation of the predicate p_2

With a similar path analysis, blocks of code that are never entered can be detected and eliminated leading to a simpler but more efficient transaction program, or an *indication to the designer that an error had been made*

5. Postconditions

It would be ideal if, from the subgoals remaining after a failed proof attempt, we could always construct the exact tests that the user had left out Unfortunately this is difficult because a simple syntactic transformation (a user error) can lead to a vastly different (semantically distant) program Hence the errors are difficult, if not impossible, to guess Even the tests arrived at through the use of equality rewrite rules[7] are not sacrosanct — while the conjuction of such tests do guarantee the safety of the current transaction "corrected" by their inclusion, the resulting "fixed" transaction may be semantically quite distant from the author's intentions Further, it is possible for the user to write a transaction that is *safe*, i e , one that does not violate any integrity constraint, but owing to an error the code of the transaction program performs quite differently from the user's intentions

In other words, when errors are caught by the safety proof mechanism, attempts to fix them may not be enough, and when no errors are detected, semantic errors may still exist We will show how both of these problems can be somewhat ameliorated through postconditions either supplied by the user or computed by the system

5.1 User-supplied postcondition

Since it is impossible to read the mind of the author of the transaction, one solution could be to require the user to supply the semantics of his own transaction, perhaps by adding a *postcondition* The theorem-prover can translate the user-supplied postcondition into a predicate *semantic-result*(say) and verify the theorem

$$(\text{integrity-pred db}) \Rightarrow (\text{semantic-result } (T' \text{ db}))$$

where T' is the original transaction T modified by the inclusion of the failed subgoals as tests If it cannot prove the theorem, it cautions the designer that although fixes are available, none are satisfactory If it can prove the theorem, it concludes that the fixed transaction is at least close enough to the user's intentions to guarantee (some of) the explicit postconditions made explicit by the user

Refer to the simple job-agency schema (with the relations Persons, Companies, Jobs, Placements, and Offerings) and consider the following provably safe transaction which the user intended to write

[7] We are not talking about the tests arrived at through a process of backchaining of implications described in section 2 4, those are clearly no more than *sufficient* conditions

```
transaction fire (company, firee, job   number) ,
preconditions
  firee in persons pid ,
  [firee, job, company] in placements ,
begin
  update p in persons where p pid = firee
    by [placed  = false ],
  remove x from placements where x pid = firee,
  if [company, job] in offerings [cid, jid]
    then update o in offerings
      where o cid = company and o jid = job
      by [number-of-positions  = number-of-positions + 1]
    else insert [company, job, 1] into offerings
end,
```

Unfortunately, the user wrote the following "bad" transaction in which he/she left out all the preconditions

```
transaction fire4 (company, firee, job   number) ,
preconditions
begin
  update p in persons where p pid = firee
    by [placed  = false ],
  remove x from placements where x pid = firee,
  if [company, job] in offerings [cid, jid]
    then update o in offerings
      where o cid = company and o jid = job
      by [number-of-positions  = number-of-positions + 1]
    else insert [company, job, 1] into offerings
end,
```

The Safety Theorem proof attempt terminates with the following message

```
The Safety Theorem
leaves these subgoals unresolved

(member company (project companies cid))
(member job (project jobs jid))
```

Now suppose that the transaction was modified by adding the above subgoals as tests The resulting transaction does not have the crucial precondition

[firee, job, company] in placements,

as a result, though the resulting transaction is *safe*, it is semantically unsafe This is because the update of the persons and placements relations and that of offerings are totally uncoupled — consequently if the input data is wrong, i e , the firee-id has no relation to the company and job-ids, [8] it is possible to increase the offerings for that company-job pair and make no change to the persons and placements relations or make changes to them that are unrelated to those in the offerings relation

If the user had supplied a useful postcondition, for example

(not (member [firee, job, company] placements/new))[9]

the theorem-prover would have been unable to prove this theorem, and the user would have been cautioned Thus postconditions can be used to test if the "fixed" transaction is close to the one intended by the user

5.2 Generating the intent of the designer

In case asking the user to write down explicit and intelligent postconditions for each transaction may be too demanding, we can take another tack and let the system compute some effects of the transaction (fixed or otherwise) The subsequent announcement of these postconditions might[10] lead the user to discover that the actual code has an effect different from what he/she intended

Of course this is a difficult problem since it is not even known which are the crucial things to compute We currently attempt only to find which relations (among the ones mentioned in the transaction program) are modified and further to determine if there is any change in cardinality of those relations

[8] If firee $\notin$ persons or firee placed = false, to begin with, then it follows that firee $\notin$ placements, and hence the update of the persons relation and the placements relation will have no effect However the offerings relation would be updated Again, if firee $\in$ persons and firee placed = true, the persons relation would be updated correctly, but the wrong company-job pair which did not exist in offerings so far, could get inserted

[9] The /new is a syntactic notation for referring to the updated state of the database

[10] In this case the user correleates these announcements with his/her knowledgeable anticipation

For example, in the above examples, the system performs as follows In transaction fire4, the system reports

```
1  modification of persons relation
   if for some x  (pid x) = firee
2  modification of placements relation
   if for some x  (pid x) = firee
3  modification of offerings relation
   unconditional
4  decrease in cardinality of placements relation
   if for some x  (pid x) = firee
5  increase in cardinality of offerings relation
   if for all x
   (not (cid x) = company) or (not (jid x) = job)
6  no change in cardinality of persons relation
```

Condition 1,2, and 4 should warn the user that there is something wrong

Consider now a transaction that violates no integrity constraints, but is one in which the user left out a subtle update

```
transaction fire1 (company, firee, job  number) ,
preconditions
  firee in persons pid ,
  [firee, job, company] in placements ,
begin
  update p in persons where p pid = firee
    by [placed  = false ],
  remove x from placements where x pid = firee,
  update o in offerings
    where o cid = company and o jid = job
      by [number-of-positions  = number-of-positions + 1]
end,
```

In this transaction, the user forgot the case in which there is no offerings tuple for the given job to start with, and needs to be inserted In this case the system reports

```
1  modification of persons relation
   unconditional
2  modification of placements relation
   unconditional
3  modification of offerings relation
   if for some x
   (cid x) = company and (jid x) = job
4  decrease in cardinality of placements relation
   unconditional
5  no change in cardinality of offerings relation
6  no change in cardinality of persons relation
```

Condition 3 is a good clue (also condition 5 to some extent) for the user that the *safe* transaction is not quite what he/she had in mind

6. Conclusion

We have discussed several forms of feedback that can be provided to the designer of a database transaction by a system using mechanical analysis and a set of theorems The theorems constitute a database theory based on a set of abstract data type axioms and recursive functions Special forms of theorems and processing are required for the different kinds of feedback

We have given examples of theorems and have shown how they can be used in the generation of useful information for designers We have shown how to generate integrity constraint checks and additional updates, either of which can be used to transform an unsafe transaction into a safe transaction However, we have argued that such transformations are not always appropriate, since they may make the transaction safe but not yield the transaction intended by the designer Rather, such additions should be suggested to the designer along with other feedback such as postcondition verification or indeed system-generated postconditions The suggested additions can be accepted by the designer or used along with the postcondition feedback to indicate errors in the transaction as written

Currently we have implemented the process of back-chaining to find sufficient tests, back-chaining to find missing updates using lemmas, and the automatic computation of postconditions, the code for using meta-lemmas to find general constraints has been implemented but not extensively tested

References

[Bernstein and Blaustein 81] Bernstein, P A and Blaustein, B T "A Simplification Algorithm for Integrity Assertions and Concrete Views" Proceedings 5th International Computer Software and Applications Conference, Chicago, Nov 1981

[Bernstein and Blaustein 82] Bernstein, P A and Blaustein, B T "Fast Methods for Testing Quantified Relational Calculus Expressions" Proceedings of 1982 ACM SIGMOD Conference, pp 39-50

[Boyer and Moore 79] Boyer, R S and Moore, J S *A Computational logic*, Acedemic Press, New York, 1979

[Bunker 86] Bunker R , "A Method for Compiling Efficient Constraint Checks in Database Programming Languages", Ph D Dissertation, Computer and Information Science, University of Massachusetts, Amherst, 1986

[Henschen et al 84] Henschen, L J , McCune, W W , and Naqvi, S A , "Compiling Constraint Checking Programs from First-Order Formulas", in *Advances in Database Theory*, Vol 2, Edited by Gallaire, Minker and Nicolas Plenum Press, New York, 1984

[Hsu and Imielinski 85] Hsu, T and Imielinski, T "Integrity Checking for Multiple Updates" Proceedings of the ACM-SIGMOD International Conference on Management of Data Austin, Texas May, 1985 pp 152-168

[Nicolas 82] Nicolas, J M "Logic for Improving Integrity Checking in Relational Databases", Acta Informatica, Vol 18, no 3, Dec 1982

[Sheard and Stemple 85] Sheard, T and Stemple, D "Coping With Complexity In Automated Reasoning About Database Systems" 11th International Conference on Very Large Databases Stockholm, Sweden August 1985 pp 426-435

[Stemple and Sheard 85] Stemple, D and Sheard, T "Database Theory for Supporting Specification-Based Database System Development" Proceedings of the 8th International Conference on Software Engineering Imperial College, London, U K August, 1985 pp 43-49

[Stemple et al 86] Stemple, D , Sheard, T and Bunker, R "Abstract Data Types in Databases Specification, Manipulation and Access" Proceedings of the IEEE Second International Conference on Data Engineering Los Angeles, California February, 1986

[Stemple and Sheard 83] Stemple, D and Sheard, T "Specification and Verification of Abstract Database Types", Third Symposium on the Principles of Database Systems, Waterloo, Ontario, Canada Apr 1984

[Sheard and Stemple 86] Sheard, T and Stemple, D , "Automatic Verification of Database Transaction Safety",COINS Technical Report 86-30, University of Massachusetts, Amherst, 1986