Coping with Complexity in Automated Reasoning about Database Systems

Tim Sheard
David Stemple
Computer and Information Science
University of Massachusetts, Amherst, Ma. 01003 USA

ABSTRACT
Automated reasoning about database systems refers to using a program or programs to draw inferences about properties of systems and can be used by designers to analyze system designs, by query processors to optimize queries, and by transaction compilers or interpreters to optimize the checking of integrity constraints. Automated reasoning can also be used as *logic programming* and combined with database processing in ways that promise to be very powerful in dealing with problems currently intractable. In these efforts, complexity will be a significant problem to be dealt with. We report on experience in dealing with complexity during efforts to mechanically prove properties of database systems.

1. Introduction

Automated reasoning about database systems refers to using a program or programs to draw inferences about properties of systems and can be used by designers to analyze system designs, by query processors to optimize queries **[King 81]**, and by transaction compilers or interpreters to optimize the checking of integrity constraints **[Gardarin and Melkanoff 79, Walker and Salveter 81, Stemple and Sheard 84]**. Automated reasoning can also be used as *logic programming* and combined with database processing in ways that are currently being studied intensely, e. g., **[Yokota et al. 84]**. Automated reasoning about databases will be included in any attempt to adapt a specification-based software development paradigm, e. g., **[Balzer et al. 83]**, to databases. Complexity is a problem which plagues, or will plague, all these uses of automated reasoning. In this paper, we report on some ways in which we have learned to cope with the complexity of the problem in the context of minimizing the integrity constraints which must be checked by transactions on highly constrained databases.

Our inference engine is a Boyer-Moore style theorem prover **[Boyer and Moore 79]**, and we have based our theory on axioms defining two abstract data types, finite sets and tuples. We have built a theory by proving mechanically over two hundred theorems and have also proven mechanically the *safety*, i. e., the property of respecting the integrity constraints on the database, of several non-trivial transactions on moderately constrained databases. In this paper, we first briefly present an outline of our theory and system specification technique and then discuss the sources of complexity and methods of dealing with it in the context of proving an example safety theorem. Though our example is from the problem of proving transaction safety, (that is the majority of our experience), we believe that the problems encountered and their solutions have significant similarity to those of the other applications of automated reasoning in database systems.

2. A brief overview of the theory and system specification

We use types throughout the formalization and specification of database systems. Two abstract data types, axiomatically defined, constitute the formal basis of all our theory. A user specifies a database with a schema comprising a set of Pascal-like type definitions: domain types, tuple types, finite set types (most of which are relation types), and a database type, the last normally consisting of a tuple type whose components are relations. Each type, at any level, can have arbitrary predicates specified as integrity constraints on objects of that type, and implicitly on any higher level type which contains it. The totality of these predicates forms a single predicate stating the consistency property of database states. See **[Stemple and Sheard 84]** for further discussion of our type specification technique.

Syntax (signature)

```
emptyset: -> fsets
rest: fsets -> fsets
choose: fsets -> elements
insert: elements X fsets -> fsets
before: elements X elements -> boolean
smaller: fsets X fsets -> boolean
```

Basic axioms

```
rest(insert(e, s)) =
if s = emptyset
    then emptyset
    else if e ≠ choose(s)
              then if before(e, choose(s))
                        then s
                        else insert(e, rest(s))
              else rest(s)

choose(insert(e, s)) =
if s = emptyset
        then e
        else if before(e, choose(s))
                then e
                else choose(s)

s ≠ emptyset -> insert(choose(s), rest(s)) = s

insert(e, s) ≠ emptyset
```

Order axioms (normally hidden)

```
a = b -> not before(a, b)

a ≠ b -> before(a, b)  or  before(b, a)

before(a, b) -> not before(b, a)

before(a, b) and before(b, c) -> before(a, c)

smaller is a well-founded relation and
s ≠ emptyset -> smaller(rest(s), s)
```

(These two axioms restrict sets to finite sets.)

Figure 2: Finite set axioms.

Transactions are functions, possibly recursive, which take typed input and a database state and return an updated database. Users write transactions in a high level language called ADABTPL which is somewhat similar to Pascal R **[Schmidt 77]**. Transactions in ADABTPL are translated into functional form expressed using the operations of the two abstract data types of the system. These operations are insert, choose, rest, and the tuple constructor and selector functions. The axioms for the tuple operations of a specific tuple type are given in figure 1, and for finite sets in figure 2. The reason for using recursive functions has been to allow the use of Boyer-Moore theorem proving techniques which are heuristics-based and rely on structural induction **[Burstall 69]**.

The axioms for tuple type person with components name and age specifies the behavior of the constructor function person and selector functions name and age by the axioms:

name(person(n, a)) = n

age(person(n, a)) = a

Figure 1: Tuple axioms.

A *safety theorem* for a transaction states that if the transaction is given a consistent database and valid input it returns a consistent database. Formally, let T be a transaction function which takes as parameters a database state DB and some input I and returns a database state. Let P be the integrated constraint on the database and Q be a predicate on the legality of the parameters I. Then the safety theorem for T is

P(DB) and Q(I) -> P(T(I,DB))

A simple specification of a one relation database with a key constraint, a simple (unsafe) transaction, and its safety theorem is given in figure 3.

3. Database schema and the database integrity constraint

To illustrate our techniques we will concentrate on a single example specifying a job agency database. The example comes from **[Gerhart 83]** and the initial specification is as follows:

"Here, persons apply for positions, companies suscribe by offering positions, and companies hire candidates or fire employees. We impose the following constraints: a person may apply only once, thus becoming a candidate, losing this status when hired by a company but regaining it if fired; a company may subscribe several times, the positive number of offerings being added up; finally, only persons that are currently candidates may be hired, and only by companies having vacant positions."

As discussed in section 2, we view the database as a single object, which can only be accessed through its transactions, much in the manner of an abstract data type. The object has structure, typically being composed of an aggregation of relations, each of which is a finite set of tuples. In addition the legal states that the database may attain are restricted by the integrity constraints. These constraints can be arbitrarily complex, and just stating them adds complexity. We describe both the structure and constraints in our system using a schema language. The language is multilevel in that types are composed of previously defined (or primitive) types and inherit the constraints of their constituent types.

```
person = [name: string, age: integer]

persons-rel = set of person where Key(name)

p-database = [persons: Persons]

transaction hire (n, a, db) =
      p-database(insert(person(n, a), persons(db)))
```

The Key constraint translates into a functional form with a relation as an explicit parameter r and a column list k. The function projT is the tuple version of the relational project.

```
key(r, k) =
if empty(r)
then true
else if member(projT(choose(r), k), project(rest(r), k))
         then false
         else key(rest(r), k)
```

The safety theorem for hire is

```
key(persons(db), name) and valid(n, a) -->
   key(persons(hire(n, a, db)), name)
```

where valid simply states that n is a string and n an integer.

Figure 3: Simple specification of database and a transaction.

```
 Schema of Job-agency

Persons = Set of
[Pid: integer, Pname: string,
 Paddr: string, Placed: ('yes', 'no')]
where Key(Pid);

Jobrel = Set of [Jid: integer, Jdescription: string]
where Key(Jid);

Companyrel = Set of
[Cid: integer, Company_name: string, Caddr: string]
where Key(Cid) and Key(Company_name);

Offer = [Cid: integer, Jid: integer,
        Number_positions: integer, Comments: string]
where Number_positions > 0;

Offerings = Set of Offer where Key(Cid, Jid);

Placementrel = Set of
[Pid: integer, Jid: integer, Cid: integer]
where Key(Pid);

Job-agency =
[Persons: Persons, Jobs: Jobrel, Offering: Offerings,
 Companies: Companyrel, Placements: Placementrel]

where

 Persons . Pid Contains Placements . Pid and
 Companies . Cid Contains Placements . Cid and
 Jobs . Jid Contains Placements . Jid and
 Jobs . Jid Contains Offering . Jid and
 Companies . Cid Contains Offering . Cid and
 For all P In Persons:
     If P . Pid In Placements . Pid
        Then P . Placed = 'yes'
        Else P . Placed = 'no'
```

Figure 4: The Job-agency schema.

The schema for our example is given in figure 4. Consider the database object's type declaration, the statement which starts with Job-agency. The database as a whole is constrained by the where clause. This clause expresses two types of interrelational constraints, referential integrity and constraints about redundant information integrity (in the For-all clause.)

Of course, other types of constraints, such as relational key constraints could be part of the database type declaration, but we believe, and experience has borne out, that "factoring" out constraints and pushing them to the lowest level of the schema allows for the most efficient mechanical proof techniques. Thus, key constraints should be with their relation type declarations. (Note that it is meaningless to place them with tuple type declarations.) A key constraint limits the tuples in a relation such that at most one tuple is allowed in the relation for any value in the component being keyed.

Constraints limit the states which the database object may legally assume. As illustrated, these constraints may be placed on any component of the database, specifying intra-tuple constraints, range constraints, constraints on a single relation such as key constraints, and interrelational constraints such as referential integrity. For purposes of proving properties of the system, these constraints are integrated into a single database predicate which specifies what it means for the database to be legal. If one were forced to place constraints only on the top level database object the correct formulation of the constraints would be more complex than it needs to be. By the use of the inheritance mechanism this complex constraint can be built up in easy stages. Thus the complete integrated integrity constraint of a database includes the interrelational constraints as well as the inherited constraints, possibly inherited through many levels. For the Job-agency example, this predicate, which we will call *thedbpred*, is a function of one variable, of Job-agency type. It is given in figure 5. The predicates persons-p, jobs-p, offering-p, companies-p and placements-p are the inherited constraints from the types, Persons, Jobrel, Offerings, Companyrel, and Placementrel, respectively. The definitions of persons-p and offering-p are:

```
persons-p(r) =
   Key(r, Pid) and
   For-all P in r: P.placed = 'yes' or
                   P.placed = 'no'
```

```
offering-p(r) =
   Key(r, Cid, Jid) and
   For all O in r: O.Number_positions > 0
```

The others are defined similarly. Note that in the offering-p constraint the Number_positions constraint in the for-all clause is itself an inherited constraint from the Offerings's constituent type offer.

When types and constraints are defined in the schema, the system automatically adds their definitions to the theorem prover's memory. This information describes differing strategies to be used to prove that some object meets a constraint or the where clause of a type.

4. A transaction and its safety theorem

Consider now the example of a transaction which hires a new employee. In the Job-agency context, such an action is only valid if the person being hired, the "Hiree", is registered with the Job-agency, the Job is offered, and the Hiree is currently unemployed. These conditions can be stated explicity in the precondition of a transaction. They should be tested before the transaction changes the three relations involved to reflect the hiring. A hire transaction in ADABTPL which we will use as the basis for a safety proof discussion is given in figure 6.

```
thedbpred(x) =

x . persons . Pid Contains x . placements . Pid and
x . companies . Cid Contains x . placements . Cid
                                   and
x . jobs . Jid Contains x . placements . Jid and
x . jobs . Jid Contains x . offering . Jid and
x . companies . Cid Contains x . offering . Cid and
For all P In x . persons:
    If P . Pid In x . placements . Pid
       Then P . Placed = 'yes'
       Else P . Placed = 'no' and

persons-p(x . persons) and
jobs-p(x . jobs) and
offering-p(x . offering) and
companies-p(x . companies) and
placements-p(x . placements)
```

Figure 5: The Job-agency database integrity predicate thedbpred.

```
Transaction
Hire (Company: integer, Hiree: integer, Job:integer);

Preconditions

  Hiree In Persons . Pid;
  [Company, Job] In Offering . [Cid, Jid];
  For the P in Persons
      where P . Pid = Hiree: P . Placed = ´no´;

 Begin {Hire body}

   For the Offer In Offering
       where Offer . [Cid, Jid] = [Company, Job]

     Do  {Update offer}
     If Offer . Number_positions = 1
        then Delete Offer from Offering
        else Update Offer using
        [Number_positions = Number_positions - 1];

   {Set Placed status for hiree to yes.}
   For the P In Persons where P . Pid = Hiree
       Update P using [Placed = ´yes´];

   {Add Placement relationship.}
   Insert [Hiree, Job, Company] into  Placements;

End {Hire transaction}.
```

Figure 6: A Hire transaction for Job-agency.

The safety theorem for the Hire transaction is

```
thedbpred(db) and integer(c) and
   integer(h) and integer(j)
  —>  thedbpred(Hire(c,h,j,db))
```

This is quite a large theorem since both thedbpred and hire are complicated functions with multiline definitions touching several (if not all) of the relations making up the Job-agency database. One may be tempted to say that sheer bulk and complexity of this theorem was caused by our own folly in insisting that the integrated database integrity constraint include all constraints. Unfortunately real databases are so constrained, and any system that hopes to solve large problems must address this problem. One of reasons for the lack of use of existing constraint checking systems is the sheer expense of checking the constraints. One states only the most important constraints, since only those can be afforded. After all, what is the use of formulating complex constraints if they are not enforced.

This example was chosen since the structure of the hire function can be broken down into three parts each of which changes a single relation. This property enhances its value as an illustrative example, but is not required by the theorem prover. Let fix-offer be a function reflecting the change in the Offering relation and fix-person be a function reflecting the changes in the Persons relation. When we "open up" the definitions of Hire and thedbpred the safety theorem for transaction Hire is expanded to the form given in figure 7.

The hypothesis of this theorem reflects both the assumption that the database was consistent to start with, and that the preconditions of the transaction are met. The conclusion of the implication reflects the constraints that must be true of the updated database. Note the use of the updating functions fix-person, fix-offer, and the primitive updating function insert to represent the changed database states. Each of the 11 conjuncts in the conclusion generates a subgoal to be proven from the whole hypothesis. Several of these subgoals are trivially true since they appear unchanged in both the hypothesis and the conclusion, the term jobs-p(db . jobs) for example. These trivial subgoals are produced when the transaction does not "touch" the object being constrained. This useful result is a byproduct of the way we state the constraints as a single unified predicate, and the way the safety theorem is formulated and expanded. The remaining subgoals are then considered one at a time. For each one we get to assume the whole hypothesis. In this example the predicates stating the correctness of the input values h (Hiree), c (Company), and j (Job) are not shown.

5. Complexity in the proof process

In the previous section we discussed how a theorem stating the safety of a complex transaction in a system with complex integrity constraints can be generated from a schema and a transaction specification and how the theorem can be broken down into smaller more manageable pieces amenable to automated proof techniques. In this section we

```
db . persons . Pid Contains db . placements . Pid and
db . companies . Cid Contains db . placements . Cid and
db . jobs . Jid Contains db . placements . Jid and
db . jobs . Jid Contains db . offering . Jid and
db . companies . Cid Contains db . offering . Cid and
For all P In db . persons:
    If P . Pid In db . placements . Pid
        Then P . Placed = 'yes'
        Else P . Placed = 'no' and
persons-p(db . persons) and
jobs-p(db . jobs) and
offering-p(db . offering) and
companies-p(db . companies) and
placements-p(db . placements)  and
h in db . Persons . Pid and
[Company, Job]  In  db . Offering . [Cid, Jid] and
For the P in db . Persons
    where P . Pid = h: P.Placed = 'no'

                    —>

fix-person(db . persons,h) . Pid Contains
    insert(placement(h,j,c), db . placements) . pid and
db . companies . Cid Contains
    insert(placement(h,j,c),db . placements . Cid and
db . jobs . Jid Contains
    insert(placement(h,j,c), db . placements) . Jid and
db . jobs . Jid Contains fix-offer(db . offering,c,j) . Jid and
db . companies . Cid Contains fix-offer(db . offering,c,j) . Cid
                    and
For all P In fix-person(db . persons,h):
   If P . Pid In insert(placement(h,j,c), db . placements) . pid
       Then P . Placed = 'yes' Else P . Placed = 'no' and

persons-p(fix-person(db . persons,h)) and
jobs-p(db . jobs) and
offering-p(fix-offer(db . offering,c,j)) and
companies-p(db . companies) and
placements-p(insert(placement(h,j,c), db . placements))
```

Figure 7: The safety theorem for transaction Hire on Job-agency.

discuss methods of reducing and dealing with the remaining complexity.

The size and complexity of safety theorems flows from two causes. First, the integration of the database predicate into a single predicate causes a large number of hypotheses to appear in the theorem. Unfortunately this is a necessary complication. The simple approach of showing for each conjunct, P_i, of the integrated predicate that $P_i(db)$ –> $P_i(T(DB))$ works for only the simplest constraints. For example, consider trying to show that if the Placements relation is keyed, then the relation insert(placement(h, j, c), placements) is also keyed.

Here placement(h, j, c) is the construction of a placement tuple from the values h for pid, j for jid, and c for cid. A proof of this follows from the precondition that the Placed attribute of h is 'no' and from the completely unrelated hypothesis that every Person who is also in Placements has his Placed attribute set to 'yes'. Thus, throwing away all hypotheses but the particular one the conclusion stems from is not a viable alternative. On the other hand, keeping all of the hypotheses around can mire the proof mechanism in great amounts of detail, much of which is unnecessary.

The second reason for the complexity of the theorem is that the changes to the database are not just simple inserts and deletes, but can be arbitrary changes to particular tuples meeting complex conditions, (like the fix-offer, and fix-person functions). In this section we discuss three methods we have used to reduce the amount of complexity arising from these causes and discuss the effects of this on obtaining proofs. The three methods are building the theory with safety proofs as a goal, stereotyping the translation of transactions, and reducing irrelevancy in theorem clauses.

5.1 Goal-directed theory

The key to handling complexity by building the "right" theory is to stereotype the common constraints and to supply the theorem prover with thorough knowledge of the interaction between the constraints and update functions. This knowledge is stored in a database of proven theorems which we refer to as "the lemmas".

Definitions:

```
project(r, &clist) =
if empty(r) then emptyset
   else insert(projT(choose(r), &clist),
                  project(rest(r), &clist))

contains(x, y) =
if empty(y) true
   else if member(choose(y), x)
           then contains(x, rest(y))
           else false
```

Lemmas:

contains-insert-rewrites:

```
contains(x, insert(a, y)) =
contains(x, y) and  member(a, x)
```

pull-insert-from-project:

```
project(insert(a, x), &clist) =
insert(projT(a, &clist), project(x, &clist))
```

Figure 8: Sample Constraint Functions and Lemmas.

For example, in our system referential integrity is specified as the containment of the projection on two relations. We have defined both containment and projection as recursive functions in our system. We have then instructed the theorem prover to prove and "remember" relevant facts about these functions. Such relevant facts involve how containment behaves under insertions, deletions, and updates, which are the very functions we expect to find in the conclusion of our safety theorems. Let us look at the definition of these two predicates and some of the "lemmas" we have proven shown in figure 8. In this definition projT is a tuple projection function, and &clist is a list of attribute names. Remember that choose selects an arbitrary element from a set, and rest is what remains once the choose is removed.

The two lemmas shown are both useful in proving invariants when the database is changed by inserts. Suppose we are trying to prove

```
contains(project(x, n), project(insert(t, y), m))
```

This is quite common if we have a referential integrity constraint between the "n" attribute of x and the "m" attribute of y. Using the lemma pull-insert-from-project the prover transforms the clause to:

```
contains(project(x, n),
        insert(projt(t, m), project(y, m)))
```

and then by the lemma contains-insert-rewrites we get the two terms:

```
contains(project(x, n), project(y, m))   and
member(projt(t, m), project(x, n))
```

the first of which is probably in the hypothesis, since it represents referential integrity in the unchanged database, and the second is a membership test. Since membership is a simpler function than contains (remember contains was defined in terms of membership) we have reduced the complexity of what we are trying to prove, and the function insert no longer even appears in the term.

We have found that by identifying, proving, and instructing the theorem prover to remember the right set of such lemmas, complexity in safety theorems is reduced and that common types of transactions can be proven safe mechanically.

5.2 Stereotyped translation of transactions

In addition to simple inserts and deletes we must be able to handle more general updates. For example, both fix-offer and fix-person are general update functions which change particular tuples in a relation which meet a nontrivial test. Although they are very different, they are also specific cases of a generic update function with the following definition:

```
update(r, p, q, f, &y) =

For the x in r where p(x, &y):
    if q(x, &y) then delete(x, r)
                  else insert( f(x, &y), delete(x, r))
```

By changing the particular function parameteters p, q, and f we can model very different specifc updates with this template. for example in fix-person, the q function is always false and thus never deletes the tuple meeting the p predicate. By proving general properties of update, then by defining fix-offer, and fix-person in terms of update, the two defined functions inherit these properties. This is a major method of reducing the complexity of the theorem prover's problem of accessing the appropriate knowledge when attempting to prove safety theorems.

This method of defining generic functions as templates which take other functions as parameters causes some problems since the particular functions passed as parameters are often quite different sometimes having different numbers of parameters in different instantiations. The notation which includes an ampersand (&) before a variable name stands for an indeterminate number of parameters. Thus in the definition of Project, &clist stands for an arbitrary number of attributes to project on. In the definition of update, the &y stands for a list of auxiliary parameters of indeterminate length which must be instantiated when a function is defined as an update. Consider the fix-person function defined as an update function.

```
fix-person(r h) =
update(r, 'named, 'falsehood, 'mark-employed, h)

where
named(x, h) = (pid(x) = h)

and
falsehood(&x) = false

and
mark-employed(x, dummy) =
person(pid(x), pname(x), Paddr(x), 'yes')
```

The &y parameter of the update function is instantiated in this function to the single variable h, which is used as an input to the call of the named function. The function falsehood returns false regardless of the number of parameters it is passed, and the function mark-employed ignores its second argument which is there only to match the number of parameters the parameter function f takes.

Now it is a theorem about update, that if the output of function parameter f is the same on the "x" attribute as on the "x" attribute of its input and the q function is always false, then the projection of the update function on the "x" attribute remains unchanged. Formally this is stated as:

if x(f(a)) = x(a) is a theorem then

(project(update(r, p, 'falsehood, f, &y), x) =
project(r, x)

is also a theorem.

We call this a meta-lemma since it is a lemma about lemmas. Whenever a function is defined in terms of update and its function parameters meet the preconditions of a theorem like the one above, then it inherits the theorem. When the fix-person function is defined, it is noted that the conclusion of the meta-lemma instantiated for fix-person is as follows

project(fix-person(r, h), x) = project(r, x)

this may be used as a lemma if

x(mark-employed(y, dummy)) = x(y)

is a theorem. At Definition time we have no value yet for the component selector x, so we cannot decide which functions x to test the hypothesis of the meta-lemma on. Once the safety theorem is produced, we scan the theorem (figure 7 for example) for instances of the conclusion fix-person(r, h) . x (using the dot notation for project) and note that Pid is a candidate value for x. The system proves that

mark-employed(y, dummy) . Pid = y . Pid

and the conclusion, with x instantiated by Pid is added as a lemma to the system. Since mark-employed never alters the pname or Paddr attributes of its input, if the terms with Pname or Paddr had been mentioned in the safety theorem, the lemmas

```
fix-person(r, h) . pname = r . pname
fix-person(r, h) . Paddr = r . Paddr
```

would also be added to the system. The theorem prover makes use of the Pid theorem when attempting to prove the subgoal in figure 7 concluding in:

```
fix-person(db . persons, h) . pid contains
    insert(placements(h, j, c), db . placements) . pid
```

Using this theorem the goal is simplified to:

```
db . persons . pid     contains
insert(placements(h, j, c), db . placements) . pid
```

Then by the contains-insert and project-insert lemmas we get:

```
h in db . persons . pid
```

Which is again one of the hypotheses and the subgoal is proven.

5.3 Reducing irrelevance in clauses

Even though many lemmas in the knowledge base reduce the complexity of the theorems being proven, theorems often contain large numbers of irrelevant terms which add nothing to a successful proof but often lead the theorem prover down blind alleys. Removing these irrelevant terms is essential to an efficient proof procedure. Consider the subgoal of the Hire transaction safety theorem which states:

```
db . persons . Pid Contains db . placements . Pid and
db . companies . Cid Contains db . placements . Cid and
db . jobs . Jid Contains db . placements . Jid and
db . jobs . Jid Contains db . offering . Jid and
db . companies . Cid Contains db . offering . Cid and
For all P In db . persons:
    If P . Pid In db . placements . Pid
        Then P . Placed = 'yes'
        Else P . Placed = 'no' and
persons-p(db . persons) and
jobs-p(db . jobs) and
offering-p(db . offering) and
companies-p(db . companies) and
placements-p(db . placements)  and
h in db . Persons . Pid and
[Company, Job]  In  db . Offering . [Cid, Jid] and
For the P in db . Persons
    where P . Pid = h: P.Placed = 'no'
            —>
placements-p(insert(placement(h, j, c), db . placements))
```

The hypothesis of the clause contains 14 terms only 4 of which are relevant. The conclusion involving placements-p is just that the placements relation is keyed on the pid attribute. The term we are trying to prove is then

```
key(insert(placement(h, j, c), db . placements), pid)
```

By applying a lemma which relates key to insert, the term is rewritten to:

```
key(db . placements, pid) and
if placement(h, j, c) in db . placements
   then true
   else projT(placement(h, j, c), pid)
        not in db . placements . pid
```

Since the first conjunct in the term above is in the hypothesis (in the guise of placements-p), we get two subgoals, one for each branch of the if statement. The then branch corresponds to the tuple placement(h, j, c) already being in the placements relation, in which case the insert adds nothing, (inserting a duplicate in the set does not change the set) and the placements relation remains keyed. The else branch is the interesting case. Here we must show that the pid attribute of the inserted tuple (which is just h) is not a member of the projection of the placements relation. We have reduced the clause to:

```
h not in db . placements . pid
```

At this point in the proof the theorem prover needs to do a proof by induction. Unfortunately, the correct recursive function to induct upon is obscured by the many irrelevant terms in the hypothesis. By throwing away all terms except the following:

```
For all P In db . persons:
   If P . Pid In db . placements . Pid
      Then P . Placed = 'yes' Else P . Placed = 'no' and
h in db . Persons . Pid and
For the P in db . Persons
     where P . Pid = h: P.Placed = 'no'
             —>
not h in db . Placements . pid
```

The correct induction can be chosen. The course of action is to do an induction on the For-all function. Throwing away the corect terms is a difficult choice for which we have as yet only the most rudimentary heuristics. If the correct irrelevant terms are not thrown away it may lead the theorem prover to attempt the wrong induction and hence miss proving the theorem. A heuristic is proposed in **[Boyer & Moore 79]** which is useful but far from complete in these circumstances. They propose partitioning the terms such that each partition contains terms which have variables mentioned in no other partition. Then, all partitions other than the one containing the conclusion term are discarded. While this works in many cases, it still leaves many irrelevant terms in the resulting clause. Throwing away irrelevant terms remains an open problem. After examining many clauses in theorems which have been intractable, we have developed intuition which allows us to select relevant clauses for use in lemmas which can be proven and then used to prove the problematic theorem. We are studying the significant problem of building at least some of our intuition (which is perhaps only effective in the domain of databases axiomatized in our style) into the theorem prover. Another approach to this problem in our context is to build an appropriate interface between the prover, which uses a purely functional form of the transactions and constraints, and the system designer who writes and thinks in higher level non-functional terms. Then a system designer could develop intuition and provide aid to the theorem prover in difficult situations. This too is being studied.

Using the techniques discussed in this section we have mechanically proven the safety theorem for Hire using our version of the Boyer-Moore theorem prover.

6. Summary

We have discussed the problems of dealing with complexity in mechanical proofs of properties of database systems. The partial solutions which we have obtained during our experience involve tailoring the theory used by the prover to the kinds of theorems which are of interest, as well as stereotyping the expression of both invariant and dynamic properties of the systems being analyzed. In our problem domain, complex transactions on highly constrained databases, large amounts of irrelevant information sometimes obscure the problem for the inference engine, and we have discussed the way in which we have had to deal with this problem, partially mechanized and partially "by hand". Though our experience is almost entirely in the domain of safety properties of database transactions and a powerful, heuristics-based prover, we believe that other domains, including logic programming with resolution-based techniques, will encounter many of the same problems and can benefit from our experince in coping with the unavoidable complexity of the problem.

References

[Balzer et al. 83] Balzer,R., Cheatham, T. E., Jr., and Green, C., "Software Technology in 1990's: Using a New Paradigm." Computer, vol. 16, no. 16, November, 1983.

[Boyer and Moore 79] Boyer, R. S. and Moore, J. S. *A Computational Logic*, Academic Press, New York, 1979.

[Burstall 69] Burstall, R. "Proving Properties of Programs by Structural Induction." Computer Journal, Vol. 12, No. 1, February, 1969, pp. 41-48.

[Gardarin and Melkanoff 79] Gardarin, G. and Melkanoff, M., "Proving Consistency of Database Transactions", Proceeding of the 5th International Conf. on Very Large Databases, 1979, pp291-298.

[Gerhart 83] Gerhart, S. "Formal Validation of a Simple Database Application." Proceedings of the Sixteenth Hawaii International Conference on System Sciences, 1983, pp. 102-111.

[King 81] King, J. J. "QUIST: A System for Semantic Optimization in Relational Data Bases", Proceedings of the Seventh International Conference on Very Large Data Bases, 1981, pp. 510-517.

[Schmidt 77] Schmidt, J. "Some High Level Constructs for Data of Type Relation." ACM Transactions on Database Systems. Vol. 2, No. 3, September 1977. pp. 247-261.

[Stemple and Sheard 84] Stemple, D. and Sheard, T., "Specification and Verification of Abstract Database Types." Third Symposium on the Principles of Database Systems, Waterloo, Ontario, April, 1984.

[Walker and Salveter 81] Walker, A. and Salveter, S. C. "Automatic Modification of Transactions to Preserve Data Base Integrity Without Undoing Updates." State University of New York, Stony Brook, New York: Tech. Report 81/026 (June 1981).

[Yokota 84] Yokota, H., Kunifuji, S., Kakuta, T., Miyazaki, N., Shibayama, S. and Murakami, K. "An Enhanced Inference Mechanism for Generating Relational Algebra Queries", Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April, 1984, pp. 229-238.