AUTOMATIC VERIFICATION OF DATABASE TRANSACTION SAFETY [1]

Tim Sheard
David Stemple

Coins Technical Report 86-30
University of Massachusetts, Amherst

ABSTRACT

Maintaining the integrity of databases is one of the promises of database management systems. This includes assuring that integrity constraints are invariants of database transactions. This is very difficult to accomplish efficiently in the presence of complex constraints and large amounts of data. One way to minimize the amount of processing required to maintain database integrity over transaction processing is to prove at compile-time that transactions cannot, if run atomically, disobey integrity constraints. We report on a system which performs such verification for a robust set of constraint and transaction classes. The system accepts database schemas written in a more or less traditional style and accepts programs in a high level programming language. Automatic verification fast enough to be effective on current workstation hardware is performed.

---

[1] This paper is based on work supported by the National Science Foundation under grant DCR-8503613.

1. Introduction

Maintaining database integrity is a difficult task. One part of this task can be accomplished by ensuring that a database obeys a set of predicates called integrity constraints after any transaction updates it, assuming that the database starts in a consistent state and is only updated by transactions. This can be done by testing at run-time all integrity constraints that could be affected by each transaction before allowing the transaction to commit. Not only could this be very expensive, but the determination of the minimal set of integrity constraints violatable by a given transaction is itself difficult. Many systems require that users specify the points at which individual integrity constraints are to be checked, for example, on specific updates or at commit time of any transaction. This, too, is problematical since users will, in even moderately complex cases, make many errors both by specifying points at which contraints could not be violated and missing points at which checks should be made.

The goal of our research is to provide a tool for developing systems which maintain the integrity of highly constrained databases, without the overhead of unnecessary tests. To achieve this goal we propose using transaction oriented databases where only *safe* transactions are allowed to update the database. A safe transaction is one that is guaranteed not to violate the integrity constraints. An automatic theorem prover is used to prove each transaction safe. Upon failing to prove a transaction safe, the system identifies what parts of the transaction and schema caused the failure. Then the system may either generate feedback to the transaction designer, or generate a set of runtime tests on the database and transaction input which can be used to guarantee the safety of the transaction.

We have built a system which is effective enough on a robust set of transactions and constraints to be useful in a database development system running in a workstation environment. The system accepts database schemas containing constraints in a relatively standard schema language and transactions in a high level programming language and proves that transactions obey the constraints stated in the schema. It also produces run-time tests as a byproduct of failed proofs. The latter capability needs improvement and is the subject of ongoing work.

The major system components are:

- A theorem prover used to build a formal theory about database systems. This theory is based on a functional model of tuples, finite sets, and natural numbers. The theory is in the style of Boyer and Moore **[Boyer and Moore 79]**, but uses a novel axiomatization of finite sets and includes higher order functions **[Stemple and Sheard 83, Sheard and Stemple 85, Stemple and Sheard 85]**.

- A schema translator used to extend the theory to a particular database by generating specific knowledge about an application from the structures and constraints contained in the database schema. The classes of constraints currently expressible in our schema language include functional dependencies, inclusion dependencies, aggregate constraints, intersection dependencies, and inter-relational redundancies, all specifiable on arbitrary subsets and derivations from a database.

- A transaction safety verifier which uses the general knowledge about database systems embodied in our database theory along with the specific knowledge generated from a schema to verify the safety of each transaction in a system. Our transactions are constructed from arbitrary conditional combinations of simple and complex updates of multiple relations. Complex updates are modelled as generic update functions which take functions as parameters to make them specific. Knowledge about the generic updates (and generic constraints which are handled in the same manner) is stored in special second order theorems, called *meta-lemmas*. Meta-lemmas are used to reason about the generic constraints and updates during the safety proof. Much of the power and efficiency of our system derives from the use of meta-lemmas.

- A test generator which provides feedback for unsafe transactions. Feedback can consist of simply identifying a constraint that cannot be proven or in some cases a sufficient set of run-time tests that could

be added to the transaction to make it safe.

Each of these components except the last is discussed at length in succeeding sections of this paper. Both the class of constraints and the class of transactions are expandable by adding to our theory new function definitions and proving the proper theorems. But this expansion of the constraint and update set is currently a job for an expert, though it is possible that it could be accomplished by database system designers when additional tools are built.

Our work is related to other work on mechanical proofs of transaction safety **[Gardarin and Melkanoff 79]**, **[Henschen et al 84]**, **[Casanova and Bernstein 80]** and **[Casanova and Bernstein 81]**. It is also related to work on simplifying constraint tests **[Bernstein and Blaustein 81]**, **[Bernstein and Blaustein 82]**, **[Walker and Salveter 81]**, **[Hsu and Imielinski 85]**, **[Nicolas 82]**. A major difference between our work and that cited is that our work makes use of a general purpose, computational logic theorem prover, and thus is based on (higher order) recursive functions rather than on first order predicate logic. Our work also deals with different (generally larger) classes of integrity constraints and transactions. Section 2.4 discusses these differences. The system represents, to the best of our knowledge, the first implementation of a transaction safety verifier which handles a realistically robust set of constraints and transactions entered in traditional schema and transaction program form. Timings on a workstation indicate that the system performs well enough to make it an effective tool for database system designers.

In the rest of this paper we first define the range of constraints and updates currently managed by our system and contrast these with previous work in the area. We then present a sample schema and four transactions and show how the

transactions are proven safe. Following this we discuss the theory and proof techniques behind the tools we have developed. We then state our conclusions and discuss future work to be done.

2. Constraint and Transaction Classes

In this section, we present the set of constraints and the transaction construction primitives available to database system designers using our development system. Constraints are specified as integral parts of a schema and the update constructs are used to build complex transactions. The schema and transaction languages are parts of our database system specification language. We first discuss salient features of this specification language using a concrete example for illustrative purposes and then present the individual constraint mechanisms and update primitives currently allowed.

2.1 The Schema Language

We call our specification language the Abstract Database Transaction Programming Language (ADABTPL which we pronounce adaptable.) It is used to write schemas and transactions. A database system specification in ADABTPL contains three sections. The first is a type definition section wherein abstract data types are constructed using tuples, lists, and finite sets along with derivation by arbitrary predicates in *where* clauses attached to each type definition. These types are used to build a database type (the second section) and to constrain transaction input. The second section describes the database object itself. The database object can also be restricted by a predicate stating the integrity constraints on the database.

The first and second sections constitute the database schema. The third section describes the set of transactions to be run on the database. Appendix 1 contains a complete example.

The database object can become constrained in two ways: directly by placing constraints in the *where* clause of the database type definition, or indirectly by inheriting constraints from the types of its components. Normally the database is defined as a tuple whose components are relations. In this case, inter-relational constraints are specifiable only in the direct method as nowhere else in the schema do two entities of type relation (i.e. a set of tuples) appear. Relational and domain constraints are more flexible as they can originate either directly in the database type where clause or by inheritance via a component type.

This generalized type definition method has some interesting consequences. First, there are several places a particular constraint can be placed in the schema. For example, suppose R is a set of tuples of type T, then the constraint that all tuples in R meet predicate P could be stated by either placing the constraint **For all x in R: P(x)** in the where clause of the relation type, or by placing **P(T)** in the where clause in the type definition of T (we use type names, e. g., T, as variables for instances of the type in where clauses used to limit the type). Similarly, relational constraints can be placed either in a relation type definition or in the where clause of the database type definition in which a database component is declared to be of that relation type. This flexibity allows the designer to attach universal constraints as low down in the schema as possible (to avoid unnecessary repetition) but also allows the designer to have two relations in the database that are specializations of the same type, differing only in their constraints. This is useful in

formally specifying certain semantic knowledge such as generalization hierarchies as well as in defining complex types which do not appear in the database itself but are used as the types of input to complex transactions.

ADABTPL provides special constructs for expressing some of the relational algebra operators. We use the dot notation to express both tuple and relational projection. If X is a tuple object with a component called name then **X.name** is the name component of X. If X is a relation then **X.name** is the set of all the name components in X. We indicate the projection of more than one component by listing them in square brackets. For example: **X.[name,age]**.

We indicate relational selection with the **(all x in R where P(x))** construct. This selects only those tuples in R where the predicate P holds.

We now introduce a particular database to illustrate the use of our system. 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. Skills are required for certain jobs, and people have abilities that satisfy skill requirements. An entity relationship (ER) diagram for the database is shown in figure 1. We will 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.

Entities and Relationships with underlined attributes are keyed by that attribute. The system should enforce that no more than one tuple with the same underlined attribute value ever exists in the database at one time.

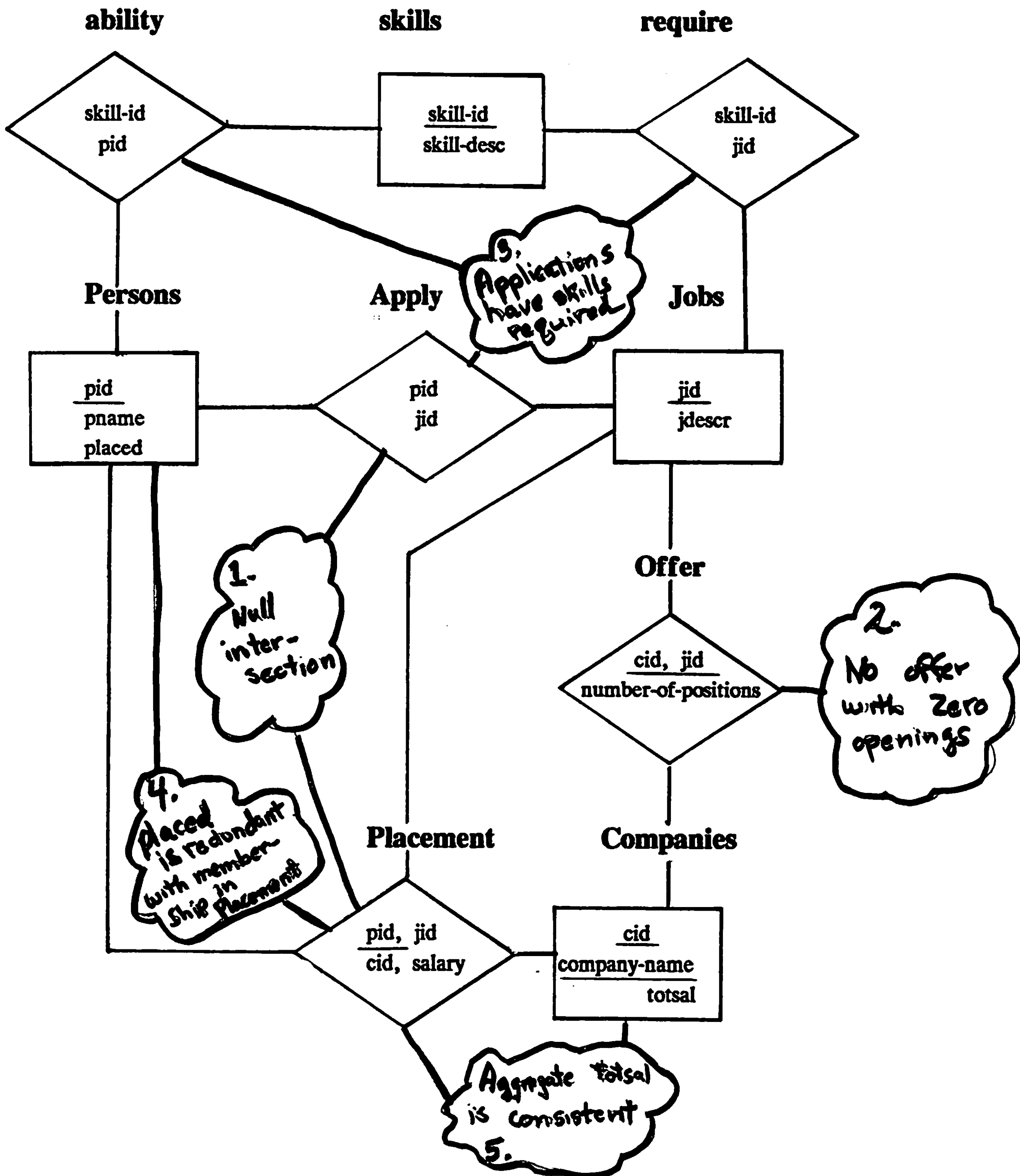

Figure 1: ER Diagram for Job Agency Database.

The diamonds of the ER diagram represent relationships. We will implement these in our ADABTPL schema as relations of foreign keys. We will need to enforce referential integrity. For example, there should never appear a **[pid, jid]** pair

in the Applications relation in which pid does not appear as a person in the Persons relation, or jid does not appear in the Jobs relation.

The relational schema given in appendix 1 for this database system explicitly states these constraints. We also constrain this system with constraints which are not implied by the ER diagram. The "clouds" in figure 1 indicate relational and inter-relational constraints which we also want the system to enforce. These constraints are as follows.

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. All persons who are applying for a job should have the skills required for the job they are applying for.

4. 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 placement 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.

5. In a similiar 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 placement relation.

2.2 Individual Constraint Mechanisms

Database integrity constraints fall in four broad categories.

1. Domain constraints which limit the values a particular component of a tuple may take on, independently of the values of other components of the tuple.

2. Tuple constraints which limit the combinations of components constituting valid tuples of a type.

3. Relational constraints which limit the value a set of tuples can take on. The *key* constraint and universal and existential quantification over the elements of the relation are examples of this type.

4. Inter-relational constraints which limit the value a relation can take on by relating it to some other relation or relations in the database. Inclusion dependencies, referential integrity, intersection dependencies, and redundancies are examples of this class of constraint.

All these kinds of constraints are expressible in our schema language using the constraint primitives presented below.[1]

We now look at the complete set of constraint mechanisms available to schema designers. Remember that designers have the option of placing many of these constraints at several levels.

2.2.1 Arithmetic Constraints

Currently the system reasons about the natural numbers. It has definitions and knowledge about the following numeric relations: equality, less-than, greater-than, less-than-or-equal, greater-than-or-equal, and non-equality. In addition it knows about the arithmetic operators addition, subtraction, multiplication, and division. In the schema language these operations are indicated by the use of the symbols, =, <, >, <=, >=, <>, +, -, *, and /, respectively. An example of an arithmetic constraint as a domain constraint follows. Remember that arithmetic constraints could be used inside universal or existential quantification constraints, or in direct database constraints. All examples in this section are taken from the job-agency schema found in appendix 1.

offer: [cid: number; jid: number; number-of-positions: number]

---

[1] We do not currently handle transition constraints, though our proof techniques should work as effectively on verifying that transactions obey these constraints as they do in verifying database invariants.

where number-of-positions <> 0 ;

The offer tuple defined here contains two foreign keys, the company identifier, cid; and the job identifier, jid. The number-of-positions field identifies how many positions company cid has open for job jid. Note that a valid offer tuple can never have zero positions open.

2.2.2 Set Membership

We allow constraints and tests which test for set membership (or non membership). In our schema and transaction languages we use the infix operator *in* to denote set membership. For example, the Hire transaction of appendix 1 includes the precondition

hiree in persons.pid

This states that in order to hire someone he must be a known entity in the database. This is tested by showing that his person identifier is a member of the projection of the Persons relation on the pid column.

2.2.3 Set Containment

Set containment is useful in stating referential integrity and other inclusion dependencies. We denote set containment using the prefix operator *contains*. By combining projection on sets and containment we can define referential integrity. For example, to declare that all those applying for jobs must be subscribing persons we state

contains (person.pid, applications.pid)

More complex inclusion dependencies can also be stated with contains. The constraint that all persons applying for a job must have abilities which include all skills

required for that job can be stated as

```
for all a in applications:
        contains( (all x in abilities where a.pid = x.pid). skill-id,
                  (all y in requirements where a.jid = y.jid). skill-id )
```

Note the use of the *all x in R where P(x)* construct which denotes set selection. We use the dot notation to denote set projection even when the set being projected is an expression rather than a variable.

2.2.4 Universal and Existential Quantification

The above constraint is also an example of universal quantification. We specify quantification over sets as

For all x in R: P(x)

For some x in R: Q(x)

where *P* and *Q* can be any predicate (even involving another quantifier.) In addition to the bound variable x, the predicates P and Q can contain unbound variables denoting the component relations defined in the database section of the schema. For example, the only unbound variables in the forall statement in 2.2.3 are abilities and requirements which are database components.

2.2.5 Null-intersection

Currently our intersection dependencies are limited to specifying that two sets have a null intersection. We use this to state that two sets do not overlap. In the job-agency example the constraint that the same person cannot simultaneously be placed and be applying for a job is stated as

null-intersection (applications.pid, placements.pid)

2.2.6 Redundancies

A database contains redundant information when some explicitly stored attributes are computable from others. This is often done to provide fast access to the redundant information. It is important that redundant parts of the database remain consistent. Our schema and transaction language provides a special construct for stating a particular kind of redundancy between two relations. Let R1 and R2 be two relations. Then

for x in R1 F(x) is-redundant-with G(x,R2)

states the redundancy that for all x in R1 the two functions F and G compute the same values when applied to (x) and (x,R2) respectively. Normally F is a component function which identifies a component of the tuples that make up R1. For example

For x in persons x.placed is-redundant-with x.pid in placements.pid

states that the placed component of all tuples in persons is either true or false, and this component agrees with the membership of the person identifier, in the projection of placements on pid. The G function is set membership denoted by the infix **in** operator.

2.2.7 Aggregate Constraints

Our schema and transaction language currently allows two aggregate functions, *sum* and *count*. These aggregate functions can exist within the normal constraint mechanisms to provide aggregate constraints. The Sum function takes two inputs, a relation and a column name we want to sum over. The column name should be numeric. Consider the constraint:

```
for c in companies c.totsal is-redundant-with
        sum( (all p in placements where p.cid = c.cid), salary)
```

which states that the totsal component of all tuples in the companies relation agrees with the sum of the salary component of all those tuples in the placements relation which are placements for the same company. Note that this constraint is a mixture of an aggregate function, a set selection and a redundancy, and as we shall see, is handled quite naturally by our system with no special mechanisms for dealing with aggregates.

Other aggregate functions such as *max* and *min* could be handled easily in a similar manner by extending the knowledge base of the sytem with function definitions and lemmas about them. The aggregate function *average* will have to wait until the system is extended to handle real numbers.

2.3 Updating Mechanisms

A transaction verification system should handle a diverse range of updating constructs, powerful enough to easily and succinctly express real database changes, yet which interact "nicely" enough with the constraint mechanisms so that safety properties of good transactions can be proven swiftly. It is easy to see that simple *inserts* and *deletes* which most previous work has dealt with are insufficient or inadequate for this task. At present our system provides, in addition to simple inserts and deletes, four complex updating constructs. We have developed our theory of constraint interaction with these six types of updates to such a degree that we can prove transactions safety theorems quickly enough that a transaction designer is encouraged to experiment, rather than be bored by long tedious proofs. The system can accommodate additional updating constructs provided suitable knowledge about their interaction with our constraints can be discovered, proved, and entered into the

system. Several new constructs are presently being evaluated for inclusion in the system. Currently this is a job for system experts. Building tools to enable database system designers to successfully add new or unique updating constructs is currently an area of research.

2.3.1 Simple Inserts and Deletes

Insertion into (or deletion from) relations is handled by building a tuple using the tuple constructor (square brackets [ ]) and inserting (or deleting) the tuple from the required relation. For example

insert [id, name, false] into persons

inserts a person with person identifier component, id; pname component, name; and placed component, false.

delete [id, name, false] from persons

would delete that tuple from the persons relation. Since these are true set operations the insert will add a tuple to a set only if it is not already in the set (i.e. no duplicates are allowed.) If a tuple is already in a set and it is inserted the insert does not change the set. In a similiar manner, **delete x from R** has no effect whenever x is not in R.

2.3.2 Remove

Often the exact tuple (or tuples) to be removed from a relation are not fully known. That is, we do not know all the components of every tuple which we want to delete. In this case the *remove* construct is useful. The remove construct deletes all tuples meeting a certain predicate from a relation. For example,

remove p from applications where p.pid = hiree

deletes all tuples from applications where the pid component has the same value as the variable hiree.

2.3.3 Multiple Inserts

Another common type of update is the insertion of several tuples at once into another relation. Often when this is the case, all the tuples share some common components but differ on some key component. We model this type of update with our **for each** construct.

for each x in R1 insert F(x) into R2

Here R1 is some set (or relation). We take each tuple in the set, transform it by the function F and insert the transformed tuple into R2. Since F is an arbitrary function it could be the identity function. In this case the for each construct models set union. Normally F constructs tuples which match the type of the target relation R2 from tuples of the type of the source relation R1. In the job-agency database when a new person subscribes to the job-agency service we would like to add to the Abilities relation a set of tuples matching each of the the new subscriber's skills with his person identifier. Let *pskills* be a set of skill-ids for the person with person identifier *id*. We do this as follows

for each s in pskills insert [s, id] into abilities

2.3.4 Update

When one wants to change some components of selected tuples in a relation the *update* construct is the appropriate choice. The format of the update construct is

update x in R where P(x) by [comp1 := F(x), comp2 := G(x), . . .]

P is a predicate function on tuples which selects those tuples to be updated. The

construction inside the square brackets tells how to construct the new value the tuple is to be replaced by. Components not mentioned in the square brackets are left unchanged. For example

```
update p in persons where p.pid = hiree by  [ placed := true ]
```

changes only those tuples in persons whose pid component is the same as the value in the variable hiree. The changed tuples are identical to their old values except that their placed components are now "true".

2.3.5 Selective Updating or Deleting

Our last updating construct allows one of three possible outcomes to be applied to each tuple. Each tuple can be 1) left unchanged, 2) deleted, or 3) have some of its components changed as is done in the normal update. We specify the use of this construct with the following format.

```
update x in R where P(x) by
        if Q(x) then delete x
                  else [ comp1 := F(x), comp2 := G(x) ...]
```

Each tuple for which P is not true is left unchanged. Tuples for which both P and Q are true are deleted. Tuples for which P but not Q is true are updated. For example, when a person is hired, it is necessary to decrement the number-of-positions available in the offerings relation by one, since one fewer positions now exist. If there was only one position available the offering tuple should be deleted since the integrity constraint on the offer type states that no offer should have a number-of-positions component of 0.

```
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 ]
```

Transactions are constructed from any number of the updates described above in some order and nested in arbitrary if-then-else structures. A transaction may optionally contain an arbitrary predicate on database and input as a precondition.

2.4 Relation of Our Constraints and Update Mechanisms to Previous Work

All previous formal work on constraint maintenance has been done in the context of the relational model. Our work has also concentrated on the relational model, but is not restricted to it. (See, for example, **[Stemple et al. 1986]**.) The work of Bernstein and Blaustein deals with simple updates and integrity constraints expressed in a limited form of predicate calculus. Hsu and Imielinski continue this work by expanding the set of constraints and dealing with sets of simple updates. Gardarin and Melkanoff take a more encompassing view, reasoning about manipulations of first normal form relations using proof techniques based on the Hoare axiomatic method **[Hoare 69]** applied to programs in ALGOL 60 extended with relational operators and first order predicates. Casanova and Bernstein's work is similiar except that they use an extended form of dynamic logic.

Nicolas uses range-restricted well formed formulas in prenex normal form. He considers single insertions and deletions (an update is a deletion followed conditionally by an insertion) of tuples. He does not allow deletes of the form "delete all tuples of employee where salary > 40000" - the user needs to find all the tuples and delete them one by one. He derives conditions for simplification of the integrity constraints under single inserts and deletes.

Henschen, McCune and Naqvi allow domain independent well formed formulas - a broader class of well formed formulas than Nicolas - in prenex form, converting the well formed formulas into clauses through skolemization. They too treat single-tuple inserts and deletes as well as a relation based update which updates all tuples which meet an equality predicate. They use a resolution theorem prover equipped with special-purpose heuristics to prove the consistency of the resulting database state, failing which they suggest that since the theorem is a disjunction any subset of the remaining clauses can be taken as a sufficient test to be performed at run-time.

Bernstein and Blaustein allow range-restricted well formed formulas in the prenex form such that a quantified variable may range over only a single relation and further, only one variable may range over a given relation. They analyse only single-tuple inserts and deletes, and allow quantifiers. They analyse cases where the constraint is automatically satisfied, others where the tests need only involve the input data, and still others where either a simplified form of the constraint can be tested, or a simple test can be performed on a reduced substate of the database stored for the purpose.

Hsu and Imielinski allow range-restricted well formed formulas in prenex form where a quantified variable may range over a single relation. Thus they allow co-range constraints where two or more variables may range over a single relation. They consider multiple inserts, and deletes by analysing the pattern of quantifiers in the prefix of the constraints. Their algorithm attempts to find such patterns that allow a decomposition of the universal and existential quantifier into a conjunction and disjunction respectively of simpler constraints. They too imply that some reduced

substates of relations may be stored to remember the "existed" information for quick computation.

Gardarin and Melkanoff express the constraints in first-order predicate calculus. Their transactions are non-trivial ALGOL-like programs with assignment statements that take care of inserts, deletes, and updates, as well as conditional statements and predicates to select specific tuples of the database. They show how Hoare's axiomatic methods can be used to verify that consistency is maintained, but their procedures are not automated.

Casanova and Bernstein also consider first order predicate calculus as the language for integrity constraints. They model transition constraints as a pair of formulas. They extend Dynamic Logic by adding axioms to model a data manipultion language enriched with relational assignment, random tuple selection, and aggregate functions and use it to prove the invariance of the constraints under the transaction programs. No steps towards mechanisation of such proofs are evident.

3. Example Schema and Transactions

In this section, we discuss the sample schema from the database system described in the ER diagram of figure 1. We also define a few transactions and use them to illustrate how a designer would use our verification system. The database consists of nine relations (one for each entity and relationship) and is constrained by twenty-one integrity constraints, comprising nine referential integrity constraints implied by the ER diagram, seven key constraints, and the five constraints described in section 2.1. The schema and transaction definitions are shown in appendix 1.

The updates are complex, one updating five relations. We believe that the transactions are of an average complexity found in real, non-toy systems and would be typical of complex transactions even on databases with much larger numbers of relations and constraints. Because of the manner in which *inertia* (the property of unupdated parts of the database continuing to obey local integrity constraints) is handled, the size of a database's structure does not unduly increase the time of a verification. Since the verification is a transaction compile-time operation, the size (extent) of the database does not affect the speed of verification at all.

We now discuss each transaction and describe its complexity and the time needed to to prove it safe.

```
transaction subscribe(id:number; name: string; pskills: set of number);
preconditions
    id not-in persons.pid;
begin
insert [id, name, false] into persons;
for each s in pskills insert [s, id] into abilities
end;
```

The simplest transaction is the Subscribe transaction. It models the event of a new person subscribing to the job agencies services. The transaction takes as input a person-identifier, a subscriber name, and a set of skills which the subscriber has mastered. It adds a new person tuple to the person relation, and a set of [skill, person-identifier] pairs to the abilities relation. It is necessary to test the precondition that the new person does not already exist in the database, or the key constraint may be violated. If the transaction were defined with the precondition omitted the result of the safety proof would be exactly this test as an unresolved subgoal. This illustrates one segment of the current test generation ability of the system. The transaction generates 8 subgoals, and is proven in 127 CPU seconds on

a DEC VAXstation II. (Our system is written in Franz LISP.)

```
transaction apply(applicant:number ; jb: number);
preconditions
    applicant in persons.pid;
    jb in jobs.jid;
    applicant not-in placements.pid;
    contains( (all a in abilities where a.pid = applicant).skill-id ,
              (all r in requirements where r.jid = jb).skill-id );
begin
insert [applicant, jb] into applications
end;
```

The apply transaction models the event of a person applying for a new job. It has four preconditions: that the applicant is a known person, the job applied for is a known type of job (note the job applied for may not be offered by any company at the current time), the applicant is not currently placed, and the applicant has the skills required for the job he is applying for. If these preconditions are omitted the test generation part of the system again discovers and reports them exactly as unresolved subgoals. The transaction generates 5 subgoals and is proven in 66 CPU seconds.

```
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;
```

The hire 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 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. This transaction generates 22 subgoals (clauses with around thirty terms each) and is proven in 255 CPU seconds.

```
transaction fire (emp,comp,jb,sal:number);
preconditions
    [emp,jb,comp,sal] in placements;
begin
remove x from placements where x.pid = emp;
update p in persons where p.pid = emp by [placed := false];
update c in companies where c.cid = comp
        by [totsal := totsal - sal]
end;
```

The fire transaction removes an employed person from the placed relation. It must set to false that person's placed field in the Persons relation, and adjust the company's totsal field in the Companies relation. Note that the precondition tests only that the placements tuple, [emp,jb,comp,sal], exists. The existence of similiar tuples in the Persons and Companies relations is proven by the system from referential integrity. This transaction generates 16 subgoals and is proven in 173 CPU seconds.

4. Safety Proof Method

Users of our system input schemas like the one found in appendix 1. This schema is passed through a schema translator which translates the schema into a set of axioms and a set of function definitions for the theorem prover. These database specific axioms and function definitions constitute a database specific knowledge base. This knowledge base along with a set of theorems encoding knowledge about databases in general are used to prove a safety theorem for each transaction in the schema.

4.1 Multi-level Knowledge Structure

The database specific knowledge generated automatically from the schema (and transactions) is the top level in a many layered system. Each layer uses a particular kind of knowledge for a particular step along the road to the ultimate goal of producing safe transactions. Each layer also uses a different set of heuristics in the proofs of its theorems. The effective integration of these layers is the key to the performance levels we need to achieve to use the system as an interactive design aid. In figure 2 we present our current four level system. On each level we list briefly the type of problem that the level handles, how knowledge is encoded, and who provides the knowledge and in what form.

The first (lowest) level in our system is an axiomatization of the basic abstract data types which the theorem prover will have to reason about. Positive integers, lists, and tuples are defined by the shell mechanism of Boyer and Moore which axiomatizes inductive data structures. We use a unique axiomatization of finite sets we have described elsewhere **[Stemple and Sheard 85]**. These are the only rules

SYSTEM LEVEL	KNOWLEDGE PROVIDER TYPE OF KNOWLEDGE
Specific Database Information 1) database structure 2) integrity constraints 3) transactions	**User Provided** User-level Specification 1) traditional schema with constraints 2) transaction programming language
Theory of Relational Databases 1) updating function definitions 2) constraint mechanisms 3) update, constraint interactions	**System Expert Provided** **Meta Rules** 1) meta-lemmas 2) inheritance 3) generic functions
Basic Theory 1) membership, containment addition, append etc. 3) tuples 4) arithmetic	**System Expert Provided** **Rules/Lemmas** 1) rewrite rules 2) induction lemmas 3) generalization rules
Abstract Data types 1) sets 2) list 3) tuples 4) natural numbers	**Axioms** **Generated From Templates** **or System Expert Provided** 1) constructor selector axioms 2) induction lemmas 3) elimination rules

Figure 2: Four Level Structure of System Knowledge.

---

which are assumed a priori. All other knowledge is built from these axioms by defining functions, suggesting theorems, proving them mechanically, and causing them to be remembered (and sometimes forgotten) for future reference.

On top of this level we build a basic theory about our primitive data types. We do this by defining simple recursive functions, e. g., addition, multiplication, set membership, and list append), then proposing, proving and remembering theorems for future use. The development of the correct set of basic theorems at this level is crucial to the operation of the system. We have spent two years refining a set of approximately 150 theorems appropriate to the task of proving safety theorems using the updating constructs and constraint mechanisms discussed earlier. This level of the system is built by a system expert. Fortunately, it need only be done once and can be shared by all database specific knowledge bases. The user of the system need not know the content or form of the knowledge base at this level. For this level, proofs by induction as described in **[Boyer and Moore 79]** are the main means of proving theorems. All theorems at this (and all other levels) have been proven mechanically using our theorem prover(s).

We next build a theory of relational databases on top of our primitive theory. This level differs from the basic theory level in that we rely heavily on generic functions, our inheritance mechanism, and meta-lemmas to encode knowledge. See section 4.3 for a detailed discussion of meta-lemmas. Careful inspection of the constraint mechanisms of section 2.2 and the updating constructs of section 2.3 shows that many of the skeleton forms described require particular predicates, selection functions, or other functions to fill out the form. We define these skeletons as generic functions, which take functions as input. At schema translation time we fill in the values for these function inputs.

The next level of the system customizes the general knowledge base for a specific database schema. This is done by translating the schema into a set of axioms and function definitions that are loaded into the theorem prover's memory. The axioms generated by the schema translator fall into the following categories.

1. The specification of the abstract data types defining the structure of the database. These abstract data type axioms are instantiated from skeleton axioms describing arithmetic, tuples, finite sets, and lists, and are generated from the specification of the component types of the schema.

2. Static properties of the database stated by the integrity constraints. Since the database always obeys the integrity constraints, axioms are entered into the theorem prover describing this knowledge in a form useful to the theorem prover.

3. Specific instances of generic theorems. We have many theorems which describe the properties of generic functions. For example:

    for a tuple with constructor c and first selector s1
    (equal (s1 (c a b)) a)

    for a relation r of such tuples
    (implies (member x r) (member (s1 x) (project r s1)))

   When we translate the schema the instantiations of the functional inputs to the generic functions become known. We can thus instantiate these theorems. For example, if *Person* is a tuple constructor function with selector functions *name* and *age*,[1] the following specific theorems with functional variables c and s1 instantiated as person and name, respectively are generated:

    (equal (name (person a b)) a)

    (implies (member x r) (member (name x) (project r name)))

   Similiar theorems are generated for age.

The schema translator also generates a set of function definitions. The functions are defined in a functional language which allows polymorphic data types and functions

[1] Knowledge generated by the schema translator, in fact all knowledge at every level, is stored and processed in the same LISP-like language used by Boyer and Moore, with higher order functions and polymorphic types added.

as first class objects. The functions generated fall into the following classes.

1. The where clause of each type statement generates a predicate definition which tests if a data entity (of the correct structure) also meets the where clause. For example, from the offer tuple definition the predicate offer-p is generated. (offer-p x) = (not (equal (number-positions x) 0)).

2. The definition of the database integrity predicate (we abreviate this as *thedbpred*) comes from two sources. The primary source is the where clause of the database definition statement in the schema. The secondary sources are the where clauses in the type declarations of the database components. These secondary sources are the inherited constraints, and may be inherited through many levels. The definition of thedbpred, thus, usually mentions schema defined predicates from class 1 above.

3. Each transaction defined generates a function. This function takes as input an object of type database, some input objects, and returns an object of type database. This function is defined in terms of calls to the generic functions which define the six updating constructs of section 3.4. Each call to one of these generic updating functions inside a transaction definition is fully instantiated with concrete functional inputs, and inherits specific lemmas from meta-lemmas wherever the functional inputs meet the meta-lemma preconditions.

Once the knowledge base is built it can be used to prove the safety of database transactions. If T is a transaction, that is a function from: **(input, database-state) to database-state** and thedbpred is the database integrity constraint that is a predicate from: **database-state to boolean**; then the safety theorem for T is:

(implies (thedbpred db) (thedbpred (T input db)))

We prove this theorem for each transaction T in the system specification.

4.2 Types of Proofs

One can see from the overview above that the system must deal with three different types of proofs.

1. Proofs in the style of Boyer and Moore to establish the basic theory of our primitive data types.

2. Meta-lemma proofs to establish the theory about generic functions which constitutes our general knowledge about database systems.

3. Proofs of individual transaction safety theorems.

Each type of proof requires a different set of heuristics and strategies. To accomodate this we have built our system in modules. A module which accepts external input we call a front-end. All our front-ends share the same internal data structures which constitute the knowledge base and use the same kernel, a term rewriting system, which includes a simplifier, function expander, and lemma applier. We have built several front-ends to our system, one to handle each kind of proof. We have also built front-ends to handle schema translation, function validation and test generation. A front-end may add only, use only, or both add and use a particuliar kind of knowledge. Figure 3 summarizes the types, sources and uses of each type of knowledge.

The function validation front-end ensures that all recursive functions are well defined (cannot recurse infinitely). It also enforces extremely strong typing. This ensures that all functions are used only where they are well defined. Function definitions contain predicates which define a total domain for that function. If the function is used (in a theorem or the definition of another function) and the domain predicate cannot be proven from the conditions which allow the function call to be reached, the use of the function is declared invalid and the theorem or function definition is not allowed.

Lemma type	Sources	Uses
generalization	basic theory	pre-induction clean-up transaction proofs
induction	shell definitions basic theory	meta-lemma proofs basic theory proofs
where-clause	schema translation function definition	transaction proofs function validation
rewrite forward-chaining & backward-chaining	basic theory shell definitions schema translation meta-lemma theory	all phases
meta-lemmas	meta-lemma definition meta-lemma inheritance	inheritance schema translation transaction proofs

Figure 3: Summary of Knowledge Source and Use.

The schema translation front-end translates the human interface language ADABTPL into the prefix functional language used by the theorem prover. The schema translator also makes note of function instantiation and adds lemmas as will be described.

The test generation front-end is the least developed of the front-ends. Its job is to use the unresolved subgoals of the transaction safety proofs and the knowledge about the database to generate sufficient tests to ensure the safety of an unsafe transaction. This is currently under development and will be the subject of a future paper.

The front-end used to add basic knowledge is essentially the Boyer and Moore theorem prover and will not be discussed here.

The meta-lemma front-end handles the proof and use of meta-lemmas. A meta-lemma describes the interaction of generic functions. It has two parts, a meta-hypothesis and a meta-body. The meta-hypothesis describes some properties of the function inputs. The meta-body is a proposition about the generic functions. We use meta-lemmas in the following manner. We attempt to prove the meta-body assuming the meta-hypothesis as a theorem. The front-end which proves meta-lemmas is quite simple. It temporarily assumes the meta-hypothesis as a rewrite rule for the duration of the proof of the meta-body. It uses the standard Boyer-Moore front-end at this stage. If the meta-body is proven the lemma is added to the meta-lemma list. The use of meta-lemmas is more complicated than their proofs. The meta-lemma list is used in the following three ways.

1. At generic function definition time, when specific functions inherit properties from the generic functions they are defined in terms of.

2. At schema translation time, to infer new facts about the specific database from the schema we are using.

3. At transaction proof time, to establish facts about instantiated functions.

The last front-end is the safety theorem prover. This requires a completely new set of heuristics different from those applied in basic theory or meta-lemma proofs. To fully understand this we need to appreciate the manner in which meta-lemmas are used.

4.3 The Use of Meta-Lemmas

Higher order functions and their associated meta-lemmas have proven to be an exceedingly parsimonious and efficent way of capturing useful generic knowledge about the domain of updating constrained databases. Higher order functions are used

to encapsulate structured updates and integrity constraint primitives. We have also used higher functions and meta-lemmas to capture knowledge common to groups of predicates sharing fundamental structure such as relational selection, set containment and universal quantification. We use meta-lemmas at various stages of knowledge generation and access. In this section, we discuss the definition of higher order functions, the form of meta-lemmas, and the use of meta-lemmas at various times. One of the most important heuristics of our system is that which determines when to generate specific lemmas in anticipation of their need versus when to generate them on demand. Unbridled anticipation, which we tried, swamps the system with lemmas, few of which are used, and consumes inordinate amounts of time in the generation process. On the other hand, relying completely on demand for generation causes certain knowledge to be missed, and incurs high search costs.

4.3.1 Generic Functions and Their Meta-lemmas

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 varaible, and that a variable that starts with an ampersand (&) is optional. The following is the form of our declaration to the function validation front-end. Choose selects an arbitrary element from a finite set. Rest is the set with the choose deleted **[Stemple and Sheard 85]**.

```
(fun (name update)

(generic-types alpha &extra)

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

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

(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. !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 result line declares that the type of the result that update returns is a set of alpha.

Meta-lemmas have the following internal form, though they may be written in somewhat simpler form when entered into the system:

(implies MH (implies H (equal L R)))

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. A rewrite lemma has a hypothesis and a rewrite rule stored as an equation and used in a left to right manner (the left-hand side is to be replaced by the right-hand side).[2] 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. Its meta-hypothesis describes properties of !p and !f, and the meta-body describes a property of update if the meta-hypothesis holds. L and R of the rewrite rule stored internally will be (key (update s !f !p &m) !c)) and true, respectively. The meta-lemma states that if !f does not change the column !c[3] in s on which a key constraint is declared, then update does not change the keyed property of s.

```
update-preserves-key: meta
(implies (equal (!c (!f x &m)) (!c x))
        (implies (key s !c) (key (update s !f !p &m) !c)))
```

---

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

[3] Remember that column names in our theory stand for selector *functions* and thus a variable standing for a column is a function variable and its use here is as a generic function (key) argument.

4.3.2 Proof-time Use of Meta-lemmas

In the process of a safety proof, whenever a generic function term is encountered, its functional parameters are concrete functions. If a meta-lemma exists whose left-hand side unifies with the term, then a specific lemma could possibly be generated. In order to generate the specific lemma we must prove the meta-hypothesis with the concrete functions. For example, if the term, **(key (update s f p &m) c))**, is encountered, it is now possible to generate a valid lemma applicable to this term if we can prove some properties of f and c. The properties that must be proven are that **(equal (c (f x &m)) (c x))**. This is a trivial example of how we use meta-lemmas during the proofs of safety theorems. It represents an example of demand driven generation of specific knowledge. The next two sections discuss examples of anticipatory generation of specific lemmas.

4.3.3 Generic Function Definition Time Use of Meta-lemmas

Generic functions can be used to gather very general knowledge about classes of functions. For example, the set operations of Membership, Containment, Existential and Universal quantification, Union, Intersection, Projection, Sum, and Selection all have common properties based on the fact that all look at each item in a set and accumulate a result. This commonality is captured by defining them all as specific instantiations of a generic mapping function. We define **s-map** (set map) recursively in terms of the set primitives choose, rest, a base object, and two generic functions, !f (the application function) and !g (the accumulator function).

```
(fun (name s-map)
(generic-types alpha beta gamma &rest)
(params (s (set of alpha))
```

```
        (!f (function (cross alpha &rest) beta))
        (!acc (function (cross beta gamma) gamma))
        (base gamma)
        (&extra &rest))

(body (if (equal s emptyset)
          base
          (!acc (!f (choose s) &extra)
                (s-map (rest s) !f !acc base &extra))))

(result gamma))
```

Using this definition as a base we define all the other functions we mentioned above. See appendix 2 for the complete definitions and some inherited lemmas. We then prove meta-lemmas about s-map and selectively inherit them, instantiated and simplified, as theorems about the derivative functions. For example, consider the definition of intersect:

```
(fun (name intersect)

(generic-types alpha)

(params (r (set of alpha)) (s (set of alpha)))

(body (s-map r
             (lambda (x y) (if (mem x y) (insert x emptyset) emptyset))
             union
             emptyset
             s))

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

Here the accumulating function is *union* and the base object is the *emptyset*. We add each element of r to the result if and only if the element is a member of s. Note the use of s in the position of the optional parameter &extra of s-map. The lambda expression in the place of the function parameter !f is a function of two variables, x which successively takes on the all the values in r, and y which is bound to s in the optional position.

Intersect inherits properties known about its generic parent, s-map. Consider the meta-lemma about s-map, s-map-insert-expands:

```
(implies (and (equal (!acc m (!acc n o)) (!acc n (!acc m o))) ; Associative
              (equal (!acc m (!acc m n)) (!acc m n))))         ; & Idempotent
         (equal (s-map (insert a s) !f !acc base &x)
                (!acc (!f a &x) (s-map s !f !acc base &x))))
```

When we define intersect in terms of s-map the meta hypothesis is instantiated as:

```
(and (equal (union m (union n o)) (union n (union m o)))
     (equal (union m (union m n)) (union m n)))
```

Applying the lemmas which state that union is idempotent and that union associates (part of our basic theory) the meta-lemma hypothesis is easily proven. Thus the meta-body is applicable to intersect. In the meta-body if we open the functions union and apply the lemmas union-insert-expands (see appendix 2) the meta-lemma body is specialized to:

```
(equal (intersect (insert a r) s)
       (if (mem a s)
           (insert a (intersect r s))
           (intersect r s)))
```

which is a useful lemma about intersect. Note the introduction of the member predicate and the absence of union in this lemma. This form has been generated by use of the term rewriting system (applying the lemma union-insert-expands) during the inheritance process. This form is more useful than the form generated directly from the meta-lemma due to the choice of theory made available to the rewrite system during this phase. The theory stored for use of the rewrite system during the various phases of processing is, in effect, expert knowledge for use in the phases.

Each of the functions defined in terms of s-map (or in terms of a function derived from s-map like for-all) inherits the s-map-insert-expands lemma if its instantiating functions meet the meta hypothesis, though the final forms of the instantiated theorem can be radically different due to the rewriting made possible by knowledge about the instantiating functional arguments. Thus, useful knowledge about the generic function s-map is inherited by the instantiated functions defined in terms of s-map. Appendix 2 contains a list of lemmas inherited from s-map.

4.3.4 Use of Meta-lemmas at Schema Translation Time

Whenever the term rewriting kernel rewrites a term, it chooses a rewrite lemma (if one exists) whose "left-hand side" unifies with the term to be rewritten. In order to avoid attempting to unify every term against the left-hand side of every rewrite rule, we index our rewrite rules by the outermost function in the left-hand term.

Because meta-lemmas allow function variables, it is possible to state a meta-lemma which cannot be indexed by a function name, since a function variable can be in the outermost function position. The only way to use such a lemma would be to attempt to apply it to *every* term, which is computationaly infeasible. Consider the meta-lemma:

```
(implies true                                   ; i.e. applies for all !p
         (implies (and (for-all r !p &a) (mem x r))
                  (!p x &a)))
```

This theorem could never be applied at proof time, since it is a theorem about !p, but !p is a function variable not a function name. We have no effective means of indexing this lemma. Instead, when a meta-lemma is added to the system, the user informs the system that the meta-lemma is "about" the higher order function (or

functions) it contains, and at schema translation time, any expressions using this function (or functions) cause the instantiation of concrete lemmas. These lemmas are indexed by the concrete functions replacing the function variable. For example, the schema fragment,

```
for all p in persons:
    p.placed = (p.pid in placements.pid)
```

translates to:

```
(for-all persons
        (lambda (x y)
                (equal (placed x)
                        (mem (pid x) (project y pid))))
        placements)
```

Note that the predicate in the ADABTPL for all statement translates into the lambda expression. This corresponds to the function variable !p in the meta-lemma. The tuple variable, p, of the for all statement corresponds to the lambda bound variable x, and the Placements relation instantiates the optional variable &a of the meta-lemma and corresponds to the lambda bound variable y. At schema translation time, we anticipate the need for a particular lemma and use the concrete values supplied for !p and &a to generate the following lemma from for-all meta-lemma above:

```
(implies (and (for-all persons
                       (lambda (x y)
                               (equal (placed x)
                               (mem (pid x) (project y pid))))
                       placements)
              (mem x persons))
         (equal (mem (pid x) (project placements pid))
                (placed x)))
```

This lemma is indexed under mem and will be available for use by the lemma choosing heuristics. This is an example of anticipatory generation of specific

knowledge and works very well in keeping both the amount of knowledge stored and search time to acceptable levels.

5. Heuristics for Safety Proofs

We will now discuss the six heuristics we use to prove safety theorems. They are:

1. Transaction and database predicate evaluation.
2. Generalization and type inheritance.
3. Elimination of inertia.
4. Subgoal generation.
5. Meta lemma instantiation.
6. Standard lemma application and function evaluation.

The first four heuristics are applied in the order listed. The last two heuristics are then applied to each subgoal alternately until the subgoal is proved or further application of these two heuristics makes no change in the subgoal. Subgoals that are left unproved are reported as unresolved and passed to the test generation front-end if the user desires. We now explain in detail each of our heuristics.

5.1 Transaction and Database Predicate Evaluation

A safety theorem has the form **(implies (P db) (P (T input db)))** where T is the transaction and P the database predicate. A database is built up from its component relations using a database *constructor* function. Given a database the component relations are accessed using database *selector* functions. The database predicate and the transaction are defined in terms of the database constructor and

selector functions. In this heuristic we open up their definitions. Since both are usually large complex functions the resulting term is very large and complex. Fortunately many of its subterms are of the form

```
(selector-i (constructor a1 a2 ... a-i ... an))
```

which reduces to a-i. To illustrate this consider the partial schema for a database named test.

```
database test : [ R1 : type1; R2 : type2; R3 : type3 ]
          where key(R1, column);
                contains(R2.column2, R3.column3);
                for all r in R3 r.column3 = "tom";

Transaction T (i1 : tuple-type1; i2 : string);
begin
insert i1 into R1;
remove x from R3 where x.column3 = i2
end;
```

These ADABTPL definitions generate the functional definitions below. Notice the use of the constructor function *test* and the selector functions *R1, R2, R3*. The result test-type is the database object's type.

```
(fun (name T)
(params (i1 tuple1-type) (i2 string) (db test-type))
(body (test (insert i1 (R1 db))
            (R2 db)
            (rem (R3 db) (lambda (x i2) (equal (column3 x) i2)) i2)))
(result test-type))


(fun (name thedbpred)
(params (db test-type))
(body (and (for-all (R1 db) tuple1-type)
           (for-all (R2 db) tuple2-type)
           (for-all (R3 db) tuple3-type)
           (key (R1 db) column1)
           (contains (project (R2 db) column2) (project (R3 db) column3))
           (for-all (R3 db) (lambda (r) (equal (column3 r) "tom")))))
(result boolean))
```

When we open up the definition of thedbpred and T in the safety theorem, (implies

(thedbpred db) (thedbpred (T input db))), we get a very large term. One of its subterms is:

```
(for-all (R3 (test (insert i1 (R1 db))
                   (R2 db)
                   (rem (R3 db)
                        (lambda (x i2) (equal (column3 x) i2))
                        i2)))
         (lambda (r) (equal (column3 r) "tom")))
```

This subterm comes from applying the last predicate of the conjunction which makes up the body of thedbpred to the expanded body of T. Each predicate in the body of thedbpred creates a similiar subterm. Note though that the set parameters of the for-all predicate can be simplified to test only the R3 portion of the database and that this subterm can be simplified to:

```
(for-all (remove (R3 db)
                 (lambda (x i2) (equal (column3 x) i2))
                 i2)
         (lambda (r) (equal (column3 r) "tom")))
```

Recapping, our first heuristic is to open up the definitions of the integrity function, thedbpred, and the transaction function. We then apply the constructor selector axioms for the database type. The application of other rewrite rules is deferred until later.

5.2 Generalization and Type Inheritance

After transaction and predicate evaluation we have a large term. This term has many subterms of the form (selector-i db). In the previous example these terms are (R1 db), (R2 db) and (R3 db). At this point the usefulness of expressing the database as a single object *db* is over. We now generalize these subterms into variables. We know the type of these variables because we know the types of the

CLIP

database components. Since *db* meets the thedbpred, we know the where clauses of the database components type apply to the new variables. Generalizing the terms (R1 db) (R2 db) (R3 db) into the variables R1, R2, R3 we get:

```
(implies (and (for-all R1 tuple1-type)
              (for-all R2 tuple2-type)
              (for-all R3 tuple3-type)
              (key R1 column1)
              (contains (project R2 column2) (project R3 column3))
              (for-all R3 (lambda ($11) (equal (column3 $11) "tom"))))
         (and (for-all (insert i1 R1) tuple1-type)
              (for-all (remove R3 (lambda ($11 $12) (equal (column3 $11) $12)) i2)
                       tuple3-type)
              (key (insert i1 R1) column1)
              (contains (project R2 column2)
                        (project (remove R3 (lambda ($11 $12) (equal (column3 $11) $12)) i2
                                 column3))
              (for-all (remove R3 (lambda ($11 $12) (equal (column3 $11) $12)) i2)
                       (lambda ($11) (equal (column3 $11) "tom")))))
```

The generalization adds the following facts about the new variables generated from the where clauses of their type definitions, i. e.,the new variables inherit their where clause type restrictions.

```
(for-all R1 tuple1-type)
(for-all R2 tuple2-type)
(for-all R3 tuple3-type)
```

where **tuple1-type** is a predicate defined by the schema translator to test the where clause of the tuple1 type definition.

The purpose of this heuristic is to add new terms to the antecedents of the clauses we will prove. It is safe to do this since we know the old database met the integrity constraints, so its components must meet their type restrictions.

5.3 Elimination of Inertia

If thedbpred contains a predicate in its body which involves only relations not changed by the transaction T, then this predicate survives unchanged in the consequent of the implication which makes up the safety proof. We call such terms inertial terms, or inertia. At this point in the proof these terms are removed from the consequent since they also appear in the antecedent of the implication. We call this process the elimination of inertia. This heuristic removes goals which would be proven trivially by the remaining heuristics.

5.4 Subgoal Generation

Each remaining term in the consequent generates a subgoal to be proven. For each subgoal we assume the facts in the antecedent of the original formula along with all the facts generated in the generalization and type inheritance phase. Each of these goals is passed individually to the last two heuristics. If all of them can be proven the safety theorem is proven.

5.5 Transaction Proof Time Use of Meta-lemmas

Each subgoal represents the interaction of one predicate of thedbpred with the parts of the database that have been changed by the transaction whose safety we are trying to prove. But these kinds of interactions are exactly the knowledge we have embodied in our meta-lemmas. In our meta-lemmas the parameter functions were represented as function variables. In our subgoals we have actual functions (most often represented as lambda expressions). We search the subgoal for a subterm which matches the body of a meta-lemma. We then try to prove the meta-hypothesis with

the actual functions replacing the function variables. Consider the subgoal which comes from the Hire transaction safety proof, which involves proving the keyed property of the offerings relation after it is updated:[1]

```
(key (update3 offerings
              (lambda (l1 l2 l3)
                      (and (equal (cid l1) l2) (equal (jid l1) l3)))
              (lambda (l1 l2 l3)
                      (equal (number-of-positions l1) (add1 0)))
              (lambda (l1 l2 l3)
                      (offer (cid l1)
                             (jid l1)
                             (sub1 (number-of-positions l1))))
              comp jb)
     (lambda (l1)
              (cross (cid l1) (jid l1))))
```

The meta-lemma: update3-preserves-keyness matches this subgoal.

```
META-HYP:
(equal (!d (!g x &y)) (!d x))

META-BODY:
(implies (key s !d) (key (update3 s !q !f1 !g &y) !d))
```

In order to apply the meta-body of the meta-lemma we must prove the instantiated meta-hypothesis which is that the two terms listed below are equal:

```
(cross (cid ((lambda (l1 l2 l3) (offer (cid l1)
                                       (jid l1)
                                       (sub1 (number-of-positions l1)))
             x comp jb))
       (jid ((lambda (l1 l2 l3) (offer (cid l1)
                                       (jid l1)
                                       (sub1 (number-of-positions l1))))
             x comp jb)))

(cross (cid x) (jid x))
```

[1] The higher order function update3 is the update form described in section 2.3.5. Its effect is on the tuples in its first argument which meet the second argument (a predicate function). These tuples are deleted if they satisfy the third argument, or if they do not, they are modified by the fourth argument.

This is easily done by applying the lambda expressions to their arguments and using the jid-offer, cid-offer constructor selector axioms. The body of the meta-lemma can now be applied to the subgoal provided the term,

```
(key offerings (lambda (l1) (cross (cid l1) (jid l1))))
```

can be proven. This term is one of the terms in the antecedent of the subgoal, since it represents the keyed property of the original offerings relation. Thus the subgoal is proven.

5.6 Standard Lemma Application and Function Evaluation

Most of the time, simply applying a meta-lemma to a subgoal is not sufficient to prove the subgoal. Many times there does not even exist a meta-lemma that matches the subgoal. If this is the case we apply our standard rewrite rules and function evaluation rules in the hope that these will either prove the subgoal or transform it to a form where a meta-lemma applies. In the job agency database we have an integrity constraint which states that no person is simultaneously in both the applications and applied relations. Since the Hire transaction changes both these relations the safety proof for hire generates the following subgoal:

```
(null-intersection (project (remove applications
                                    (lambda (l1 l1) (equal (pid l1) l2))
                                    hiree)
                            pid)
                   (project (insert (placement hiree jb comp sal) placements)
                            pid))
```

No meta-lemmas apply to this subgoal. However, the following rewrite lemmas apply to subterms of the the subgoal.

```
pull-insert-out-of-project: rewrite
(equal (project (insert a x) !d) (insert (!d a) (project x !d)))
```

```
project-rem-to-delete-project: rewrite meta
(equal (project (remove a (lambda (l1 l2) (equal (!c1 l1) l2)) n) !c1)
       (delete n (project a !c1)))
```

Once these lemmas are applied, the subgoal becomes

```
(null-intersection (delete hiree (project applicatons pid))
                   (insert (pid (placement hiree jb comp sal))
                           (project placements pid)))
```

and the following lemmas become applicable.

```
null-inter-insert-expands2: rewrite
(equal (null-intersection x (insert a y))
       (if (null-intersection x y) (not (mem a x)) false))

null-inter-means-null-inter-delete: rewrite
(implies (null-intersection x y) (null-intersection x (delete a y)))
```

The second rewrite rule can be applied only if we can prove the premise:

```
(null-intersection (project applications pid) (project placements pid))
```

But this premise is in the antecedent of the subgoal since it represents the fact that the database met the integrity constraints before the transaction started. Thus the consequent of the subgoal rewrites to:

```
(not (mem (pid (placement hiree jb comp sal))
          (delete hiree (project applications pid))))
```

Applying the lemmas mem-delete-rewrites, and the constructor selector axiom pid-placement the subgoal is proven.

```
mem-delete-rewrites: rewrite
(equal (mem a (delete b x)) (if (equal a b) false (mem a x)))
```

6. Summary

We have demonstrated that mechanical theorem proving in higher order computational logic can be used effectively on the problem of proving that complex database transactions obey complex database constraints. To accomplish this, it was necessary to

1. axiomatize a few abstract data types from which databases could be structured,
2. build a basic theory combining the theories of these abstract data types,
3. extend Boyer and Moore style theorem proving to higher order functions, treating types and functions (including predicates) as first class objects,
4. build different control front-ends for proving basic theorems, transaction safety theorems, and theorems involving higher order functions and conditions on their functional inputs, and
5. develop a general theory of database systems which has the right structure for guiding the safety theorem prover to effective safety proofs.

The system we have implemented allows database system designers to specify a database system using a more or less standard schema language along with a high level transaction programming language. A designer need know nothing about the internal logical form of the resulting system specification nor any details of proof procedures to use the system effectively. The schema and programming language is suitable as the source for translation into a lower level programming language and database management system.

The current system deals with a robust set of constraint constructs and transaction forms and can be extended with relative ease. Currently extensions can only be made by a system expert, not by database designers. Current research addresses the problem of database system designers extending the system's reasoning

capabilities. A basic facility for generating run-time tests for unsafe transactions has been implemented, and an improved facility is under development. Other improvements to our system include allowing user-defined abstract data types in databases, semantic query optimization, and compilation of our high level language into target database management systems.

Our verification system is an expert system in which the expert knowledge is embedded in the heuristics of the front-ends (encoded procedurally in LISP code) and in the lemmas of the different levels of the system (encoded nonprocedurally [?] in computational logic lemmas). The interpreter of the knowledge is probably more sophisticated than is common in most expert system work, though we do not claim to be familiar with the state of the art in expert systems. One of the differences between our non-procedural knowledge base, the lemmas, and that of most expert systems is that our base has been formally and mechanically verified in its entirety. The sophistication of our interpreter is only one of the similarities between our approach and that of Boyer and Moore, to whom we owe so much for inspiration as well as for concrete techniques, heuristics and guidance in knowledge base building. One of the differences between our work and theirs is our dependence on automatic generation of specific knowledge from generic knowledge. Perhaps the most important lesson that we have learned in our efforts is that building a theory for use in practical reasoning is a difficult and delicate operation, to be attempted only if sufficient patience for the long haul is possessed. This is most likey true even if the knowledge interpreter is less sophisticated than ours. In other words, writing programs to be interpreted by sophisticated knowledge oriented interpreters is even harder than writing programs for interpreters with simpler evaluation loops such as LISP

interpreters, Prolog systems, Pascal machines, or 8086 processors. However, we believe the results which can be attained by combining complex heuristics with formally verified knowledge bases to be worth the effort and achievable, in many cases, in no other way.

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.

[Casanova and Bernstein 80] Casanova, M. A. and Bernstein, P. A. "A Formal System for Reasoning about Programs Accessing a Relational Database", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3. Jul. 1980.

[Casanova and Bernstein 79] Casanova, M. A. and Bernstein, P. A. "A Logic of a Relational Data Manipulation Language." Sith ACM Symposium on Principles of Programming Languages." Jan. 1979. pp. 101-120

[Gardarin and Melkanoff 79] Gardarin, G. and Melkanoff, M. "Proving the Consistency of Database Transactions", Proceedings of the 5th International Conference on Very Large Databases, Rio de Janiero, Brazil, 1979.

[Henschen et al 84] Henschen, L. J., McCune, W. W., and Naqvi, S. A. *Advances in Database Theory*, Vol 2. Edited by Gallaire, Minker and Nicolas. Plenum Press, New York, 1984.

[Hoare 69] Hoare, C. A. "An Axiomatic Basis for Computer Programming." Communications of the ACM, Vol 12, No. 10, Oct. 1969.

[Hsu and Imielinski] 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 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

[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.

[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).

Appendix 1

In this appendix we list the ADABTPL schema of the job agency database used as an example in the main text. It has three sections, type definitions, the database definition, and transaction definitions.

```
schema of job-agency ;

type

person: [ pid: number; pname: string; placed: boolean ];

person-rel: set of person-type where  key(person-rel, pid);

job: [ jid: number; jdescript: string ] ;

job-rel: set of job-type where key(job-rel, jid);

skill: [ skill-id: number; skill-descript: string ];

skill-rel: set of skill-type where key(skill-rel, skill-id);

ability: [ skill-id: number; pid: number ];

ability-rel: set of ability-type;

requirement: [ skill-id: number; jid: number ];

requirement-rel: set of requirement-type;

company: [ cid: number; company-name: string; totsal: number];

company-rel: set of company-type
                where  key(company-rel, cid) and  key(company-rel, company-name);

application: [pid: number; jid: number ];

application-rel: set of application-type;

offer: [ cid: number; jid: number; number-of-positions: number ]
        where number-of-positions <> 0 ;

offer-rel: set of offer-type where  key(offer-rel, cid, jid);

placement: [ pid: number; jid: number; cid: number; salary: number ];

placement-rel: set of placement-type where key(placement-rel, pid);
```

```
database job-agency : [ persons: person-rel;
                        jobs:  job-rel;
                        skills: skill-rel;
                        abilities: ability-rel;
                        requirements: requirement-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, abilities.pid) ;
      contains(persons.pid, applications.pid) ;
      contains(jobs.jid, applications.jid) ;
      contains(jobs.jid, requirements.jid) ;
      for x in persons x.placed is-redundant-with x.pid in placements.pid ;
      for all a in applications:
            contains( (all x in abilities where a.pid = x.pid).skill-id,
                      (all y in requirements where a.jid = y.jid).skill-id );
      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);
```

```
transaction apply(applicant:number ; jb: number);
preconditions
    applicant in persons.pid;
    jb in jobs.jid;
    applicant not-in placements.pid;
    contains( (all a in abilities where a.pid = applicant).skill-id ,
              (all r in requirements where r.jid = jb).skill-id );
begin
insert [applicant, jb] into applications
end;


transaction subscribe(id:number; name: string; pskills: set of number);
preconditions
    id not-in persons.pid;
begin
insert [id, name, false] into persons;
for each s in pskills insert [s, id] into abilities
end;


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;


transaction fire (emp,comp,jb,sal:number);
preconditions
    [emp,jb,comp,sal] in placements;
begin
remove x from placements where x.pid = emp;
update p in persons where p.pid = emp by [placed := false];
update c in companies where c.cid = comp
        by [totsal := totsal - sal]
end;
```

Appendix 2

In this appendix we list the meta-lemma s-map-insert-expands and the specific lemmas generated from it by our definition time inheritance mechanism. In each case we give the definition of the child function in terms of s-map, and the lemma generated. After the inherited lemma's name we list how we use the lemma. Note some of the inherited lemmas are themselves meta lemmas. Note also the different forms of the instantiations of the body of s-map-insert-expands for the different specializations of s-map. These forms are the result of using the term rewriting system and knowledge about the instantiating functional arguments during the inheritance process.

```
s-map-insert-expands: meta
(implies (and (equal (!acc m (!acc n o)) (!acc n (!acc m o)))
              (equal (!acc m (!acc m n)) (!acc m n)))
         (equal (s-map (insert a s) !f !acc base &x)
                (!acc (!f a &x) (s-map s !f !acc base &x))))
```

```
(fun (name for-some)
     (types alpha &beta)
     (params (x (set of alpha))
             (!p (function (cross alpha &beta) boolean))
             (&extra &beta))
     (body (s-map x !p or false &extra))
     (result boolean)
     (inherit meta))
```

```
for-some-insert-expands: rewrite meta
(equal (for-some (insert a x) !p &extra)
     (if (!p a &extra) true (for-some x !p &extra)))
```

```
(fun (name for-all)
     (types alpha &beta)
     (params (x (set of alpha))
             (!p (function (cross alpha &beta) boolean))
             (&extra &beta))
     (body (s-map x !p and true &extra))
     (result boolean)
     (inherit meta))

for-all-insert-expands: rewrite meta
(equal (for-all (insert a x) !p &extra)
       (if (!p a &extra) (for-all x !p &extra) false))



(fun (name mem)
     (types alpha)
     (params (a alpha) (x (set of alpha)))
     (body (for-some x equal a))
     (result boolean))

mem-insert-expands: rewrite
(equal (mem a (insert a1 x)) (if (equal a1 a) true (mem a x)))



(fun (name contains)
     (types set-elem)
     (params (r (set of set-elem)) (s (set of set-elem)))
     (body (for-all s mem r))
     (result boolean))

contains-insert-expands: rewrite
(equal (contains r (insert a s)) (if (mem a r) (contains r s) false))



(fun (name union)
     (types set-elem)
     (params (r (set of set-elem)) (s (set of set-elem)))
     (body (s-map r identity insert s))
     (result (set of set-elem)))

union-insert-expands: rewrite
(equal (union (insert a r) s) (insert a (union r s)))
```

```
(fun (name select)
     (types alpha)
     (params (r (set of alpha)) (!p (function alpha boolean)))
     (body (s-map r (lambda (x) (if (!p x) (insert x emptyset) emptyset))
                    union emptyset))
     (result (set of alpha)))
```

select-insert-expands: rewrite

```
(equal (select (insert a r) !p)
       (if (!p a) (insert a (select r !p)) (select r !p)))
```

```
(fun (name intersect)
     (types alpha)
     (params (r (set of alpha)) (s (set of alpha)))
     (body (s-map r (lambda (x y) (if (mem x y) (insert x emptyset) emptyset))
                      union
                      emptyset
                      s))
     (result (set of alpha)))
```

intersect-insert-expands: rewrite

```
(equal (intersect (insert a r) s)
       (if (mem a s) (insert a (intersect r s)) (intersect r s)))
```

```
(fun (name project)
     (types set-elem new-set-elem)
     (params (r (set of set-elem)) (!p (function set-elem new-set-elem)))
     (body (s-map r !p insert emptyset))
     (result (set of new-set-elem)))
```

project-insert-expands: rewrite

```
(equal (project (insert a r) !p) (insert (!p a) (project r !p)))
```

Appendix 3

In this appendix we list the complete proof of the safety theorem for the Fire transaction. We use a verbose mode for the the first subgoal to show the use of meta-lemmas and a terse mode for the other subgoals to save space.

We will attempt to prove the conjecture:

```
(implies (thedbpred db) (thedbpred (fire1 emp comp jb sal db)))
```

Opening the functions fire1, thedbpred and applying the constructor-destructor lemmas placements-job-agency, companies-job-agency, offerings-job-agency, applications-job-agency, requirements-job-agency, abilities-job-agency, skills-job-agency, jobs-job-agency, persons-job-agency then generalizing the terms (persons db), (jobs db), (skills db), (abilities db), (requirements db), (applications db), (offerings db), (companies db), (placements db) into the variables persons, jobs, skills, abilities, requirements, applications, offerings, companies, placements we get a formula whose antecedent is:

```
(and (key persons pid)
     (for-all persons person-type)
     (key jobs jid)
     (for-all jobs job-type)
     (key skills skill-id)
     (for-all skills skill-type)
     (for-all abilities ability-type)
     (for-all requirements requirement-type)
     (for-all applications application-type)
     (key offerings (lambda ($11) (cross (cid $11) (jid $11))))
     (for-all offerings offer-type)
     (key companies cid)
     (key companies company-name)
     (for-all companies company-type)
     (key placements pid)
     (for-all placements placement-type)
     (contains (project persons pid) (project placements pid))
     (contains (project companies cid) (project placements cid))
     (contains (project jobs jid) (project placements jid))
     (contains (project jobs jid) (project offerings jid))
     (contains (project companies cid) (project offerings cid))
     (contains (project persons pid) (project abilities pid))
     (contains (project persons pid) (project applications pid))
     (contains (project jobs jid) (project applications jid))
     (contains (project jobs jid) (project requirements jid))
     (redun persons
            placed
```

```
        (lambda ($11 $12) (mem (pid $11) (project $12 pid)))
        placements)
  (for-all applications
        (lambda ($11 $12 $13)
                (contains (project (select $12
                                           (lambda ($11 $12)
                                                   (equal (pid $11) $12))
                                           (pid $11))
                                   skill-id)
                          (project (select $13
                                           (lambda ($11 $12)
                                                   (equal (jid $11) $12))
                                           (jid $11))
                                   skill-id)))
        abilities
        requirements)
 (null-intersection (project applications pid) (project placements pid))
 (redun companies
        totsal
        (lambda ($11 $12)
                (sum (select $12
                             (lambda ($11 $12)
                                     (equal (cid $11) $12))
                             (cid $11))
                       salary))
        placements)
        (mem (placement emp jb comp sal) placements))
```

The generalization adds the following facts about the new variables generated from the where clauses of their type definitions:

```
(for-all persons person-type)
(key persons pid)
(for-all jobs job-type)
(key jobs jid)
(for-all skills skill-type)
(key skills skill-id)
(for-all abilities ability-type)
(for-all requirements requirement-type)
(for-all applications application-type)
(for-all offerings offer-type)
(key offerings (lambda ($11) (cross (cid $11) (jid $11))))
(for-all companies company-type)
(key companies company-name)
(key companies cid)
(for-all placements placement-type)
(key placements pid)
```

The consequent of the formula has 16 subgoals. The functions delete, rest are disabled. No lemmas are disabled.

****

The consequent for the 1st subgoal is:

```
(key (update persons
             (lambda ($11 $12)
                     (person (pid $11) (pname $11) false))

             (lambda ($11 $12)
                     (equal (pid $11) $12))

             emp)
     pid)
```

We will attempt to use the meta-lemma: update-preserves-key

```
HYP:
(equal (*!c (*!f x &m)) (*!c x))
BODY:
(implies (key *s *!c) (key (update *s *!f *!p &m) *!c))
```

Applying the lemmas pid-person the (instantiated) meta hypothesis

```
(equal (pid (person (pid x) (pname x) false)) (pid x))
```

of the meta-lemma update-preserves-key is proven. Applying the lemmas update-preserves-key

```
update-preserves-key: meta
(implies (equal (!c (!f x &m)) (!c x))
         (implies (key s !c) (key (update s !f !p &m) !c)))
```

and referring to term 1 of the antecedent

```
1
(key persons pid)
```

The consequent is proven.

****

The consequent for the 2nd subgoal is:

```
(for-all (update persons
                 (lambda ($11 $12)
                         (person (pid $11) (pname $11) false))
                 (lambda ($11 $12)
                         (equal (pid $11) $12))
                 emp)
         person-type)
```

We will attempt to use the meta-lemma: for-all-update
Applying the lemmas person-type the (instantiated) meta hypothesis of the meta-lemma for-all-update is proven. Applying the lemmas for-all-update and referring to term 2 of the antecedent the consequent is proven.

****

The consequent for the 3rd subgoal is:

```
(key (update companies
             (lambda ($11 $12 $13)
                     (company (cid $11)
                              (company-name $11)
                              (difference (totsal $11) $13)))
             (lambda ($11 $12 $13)
                     (equal (cid $11) $12))
             comp
             sal)
     cid)
```

We will attempt to use the meta-lemma: update-preserves-key
Applying the lemmas cid-company the (instantiated) meta hypothesis of the meta-lemma update-preserves-key is proven. Applying the lemmas update-preserves-key and referring to term 12 of the antecedent the consequent is proven.

****

The consequent for the 4th subgoal is:

```
(key (update companies
             (lambda ($11 $12 $13)
                     (company (cid $11)
                              (company-name $11)
                              (difference (totsal $11) $13)))
             (lambda ($11 $12 $13)
                     (equal (cid $11) $12))
             comp
             sal)
     company-name)
```

We will attempt to use the meta-lemma: update-preserves-key

18, 30 of the antecedent the consequent is proven.

****

The consequent for the 10th subgoal is:

```
(contains (project jobs jid)
          (project (rem placements
                        (lambda ($11 $12)
                                (equal (pid $11) $12))
                        emp)
                   jid))
```

Applying the lemmas contains-project-delete-small-set,
key-means-rem-rewrites, pid-placement and referring to the terms 15,
19, 30 of the antecedent the consequent is proven.

****

The consequent for the 11th subgoal is:

```
(contains (project (update companies
                           (lambda ($11 $12 $13)
                                   (company (cid $11)
                                            (company-name $11)
                                            (difference (totsal $11) $13)))
                           (lambda ($11 $12 $13)
                                   (equal (cid $11) $12))
                           comp
                           sal)
                   cid)
          (project offerings cid))
```

We will attempt to use the meta-lemma: project-update-simplifies
Applying the lemmas cid-company the (instantiated) meta hypothesis of the
meta-lemma project-update-simplifies is proven. Applying the lemmas
project-update-simplifies and referring to term 21 of the antecedent the
consequent is proven.

****

The consequent for the 12th subgoal is:

```
(contains (project (update persons
                           (lambda ($11 $12)
                                  (person (pid $11) (pname $11) false))
                           (lambda ($11 $12)
                                  (equal (pid $11) $12))
                           emp)
                    pid)
          (project abilities pid))
```

We will attempt to use the meta-lemma: project-update-simplifies
Applying the lemmas pid-person the (instantiated) meta hypothesis of the meta-lemma project-update-simplifies is proven. Applying the lemmas project-update-simplifies and referring to term 22 of the antecedent the consequent is proven.

****
The consequent for the 13th subgoal is:

```
(contains (project (update persons
                           (lambda ($11 $12)
                                  (person (pid $11) (pname $11) false))
                           (lambda ($11 $12)
                                  (equal (pid $11) $12))
                           emp)
                    pid)
          (project applications pid))
```

We will attempt to use the meta-lemma: project-update-simplifies
Applying the lemmas pid-person the (instantiated) meta hypothesis of the meta-lemma project-update-simplifies is proven. Applying the lemmas project-update-simplifies and referring to term 23 of the antecedent the consequent is proven.

****
The consequent for the 14th subgoal is:

Applying the lemmas company-name-company the (instantiated) meta hypothesis of the meta-lemma update-preserves-key is proven. Applying the lemmas update-preserves-key and referring to term 13 of the antecedent the consequent is proven.

****

The consequent for the 5th subgoal is:

```
(for-all (update companies
                 (lambda ($11 $12 $13)
                         (company (cid $11)
                                  (company-name $11)
                                  (difference (totsal $11) $13)))
                 (lambda ($11 $12 $13)
                         (equal (cid $11) $12))
                 comp
                 sal)
         company-type)
```

We will attempt to use the meta-lemma: for-all-update
Applying the lemmas company-type the (instantiated) meta hypothesis of the meta-lemma for-all-update is proven. Applying the lemmas for-all-update and referring to term 14 of the antecedent the consequent is proven.

****

The consequent for the 6th subgoal is:

```
(key (rem placements (lambda ($11 $12)
                             (equal (pid $11) $12))
                     emp) pid)
```

Applying the lemmas key-means-key-delete, key-means-rem-rewrites, pid-placement and referring to the terms 15, 30 of the antecedent the consequent is proven.

****

The consequent for the 7th subgoal is:

```
(for-all (rem placements (lambda ($11 $12)
                                 (equal (pid $11) $12))
                         emp)
         placement-type)
```

We will attempt to use the meta-lemma: for-all-rem
Applying the lemmas for-all/placement-rel and referring to term 16 of the antecedent the (instantiated) meta hypothesis of the meta-lemma for-all-rem is proven. Applying the lemmas for-all-rem and referring to term 16 of the antecedent the consequent is proven.

****

The consequent for the 8th subgoal is:

```
(contains (project (update persons
                           (lambda ($11 $12)
                                   (person (pid $11) (pname $11) false))
                           (lambda ($11 $12)
                                   (equal (pid $11) $12))
                           emp)
                   pid)
          (project (rem placements
                        (lambda ($11 $12)
                                (equal (pid $11) $12))
                        emp)
                   pid))
```

We will attempt to use the meta-lemma: project-update-simplifies
Applying the lemmas pid-person the (instantiated) meta hypothesis of the meta-lemma project-update-simplifies is proven. Applying the lemmas contains-delete-small-set, pull-delete-from-project, key-means-rem-rewrites, pid-placement, project-update-simplifies and referring to the terms 15, 17, 30 of the antecedent the consequent is proven.

****

The consequent for the 9th subgoal is:

```
(contains (project (update companies
                           (lambda ($11 $12 $13)
                                   (company (cid $11)
                                            (company-name $11)
                                            (difference (totsal $11) $13)))
                           (lambda ($11 $12 $13)
                                   (equal (cid $11) $12))
                           comp
                           sal)
                   cid)
          (project (rem placements
                        (lambda ($11 $12)
                                (equal (pid $11) $12))
                        emp)
                   cid))
```

We will attempt to use the meta-lemma: project-update-simplifies
Applying the lemmas cid-company the (instantiated) meta hypothesis of the meta-lemma project-update-simplifies is proven. Applying the lemmas contains-project-delete-small-set, key-means-rem-rewrites, pid-placement, project-update-simplifies and referring to the terms 15,

```
(redun (update persons
               (lambda ($11 $12)
                       (person (pid $11) (pname $11) false))
               (lambda ($11 $12)
                       (equal (pid $11) $12))
               emp)
       placed
       (lambda ($11 $12)
               (mem (pid $11) (project $12 pid)))
       (rem placements (lambda ($11 $12)
                               (equal (pid $11) $12))
                       emp))
```

We will attempt to use the meta-lemma: redun-update-rem
Applying the lemmas mem-project-placement-on-pid, pull-delete-from-project, key-means-rem-rewrites, pid-placement, placed-person, pid-person, mem-delete-rewrites, project-rem-to-delete-project and referring to the terms 15, 30 of the antecedent the (instantiated) meta hypothesis of the meta-lemma redun-update-rem is proven. Applying the lemmas redun-update-rem and referring to term 26 of the antecedent the consequent is proven.

****

The consequent for the 15th subgoal is:

```
(null-intersection (project applications pid)
                   (project (rem placements
                                 (lambda ($11 $12)
                                         (equal (pid $11) $12))
                                 emp)
                            pid))
```

Applying the lemmas null-inter-means-null-inter-delete, pull-delete-from-project, key-means-rem-rewrites, pid-placement and referring to the terms 15, 28, 30 of the antecedent the consequent is proven.

****

The consequent for the 16th subgoal is:

```
(redun (update companies
               (lambda ($11 $12 $13)
                       (company (cid $11)
                                (company-name $11)
                                (difference (totsal $11) $13)))
               (lambda ($11 $12 $13)
                       (equal (cid $11) $12))
               comp
               sal)
       totsal
       (lambda ($11 $12)
               (sum (select $12
                            (lambda ($11 $12)
                                    (equal (cid $11) (cid $12)))
                            $11)
                    salary))
       (rem placements (lambda ($11 $12)
                               (equal (pid $11) $12))
                       emp))
```

We will attempt to use the meta-lemma: redun-update-rem
Applying the lemmas salary-placement, sum-delete-expands, mem-select-expands, select-delete-expands, cid-placement, cid-company, redun-means/sum3, key-means-rem-rewrites, pid-placement, totsal-company, select-lambda-special and referring to the terms 15, 29, 30 of the antecedent the (instantiated) meta hypothesis of the meta-lemma redun-update-rem is proven. Applying the lemmas redun-update-rem, select-lambda-special and referring to term 29 of the antecedent the consequent is proven.

The Safety Theorem is Proven!

The following are the meta-lemmas referred to in the proof.

```
update-preserves-key: meta
(implies (equal (!c (!f x &m)) (!c x))
         (implies (key s !c) (key (update s !f !p &m) !c)))

for-all-update: meta
(implies (implies (!p x &rest1) (!q (!f x &rest1) &rest2))
         (implies (for-all r !q &rest2)
                  (for-all (update r !f !p &rest1) !q &rest2)))

for-all-rem: meta
(implies (if (mem x *r) (if (*!c x &n) true (*!q x &rest2)) true)
         (implies (for-all *r *!q &rest2)
                  (for-all (rem *r *!c &n) *!q &rest2)))
```

```
project-update-simplifies: meta
(implies (equal (!d (!f x &m)) (!d x))
         (equal (project (update r !f !p &m) !d) (project r !d)))

redun-update-rem: meta
(implies (and (implies (and (mem x r1) (not (!p x &b)))
                           (equal (!q x (rem r2 !s &a)) (!q x r2)))
              (implies (and (mem x r1) (!p x &b) (equal (!f x) (!q x r2)))
                           (equal (!q (!t x &b) (rem r2 !s &a)) (!f (!t x &b)))))
         (implies (redun r1 !f !q r2)
                  (redun (update r1 !t !p &b) !f !q (rem r2 !s &a))))
```