INCORPORATING THEORY INTO DATABASE SYSTEM DEVELOPMENT

David Stemple
Tim Sheard
Ralph Bunker

COINS Technical Report 85-08
March 1985

INCORPORATING THEORY INTO DATABASE SYSTEM DEVELOPMENT

David Stemple
Tim Sheard
Ralph Bunker
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts

ABSTRACT

Database systems, like all models, must be constrained to represent just those states and transitions which are possible in the world they model. Database integrity constraints, transition constraints and transaction definitions specify the conformity of a database system to the real world. However, when a system is implemented these specifications must be followed, the constraints enforced, if the model is to be valid. The enforcement of database system constraints is a difficult problem to solve efficiently; this follows from the large amounts of data involved and the complexity of determining minimum required checks. We present a database system development method in which considerable theoretical support in the form of automated theorem proving is brought to bear on the integrity enforcement prolem. The theory underlying the method and the power of the theorem prover also allow the system to provide a designer feedback on the quality of schemas and transactions as well as on the behavioral implications of the system's specification.

Appeared in Information Processing
and Management, Vol. 22, No. 4,
1986, pp. 317-330.

1. Introduction

A database system, like any model, needs to be constrained in order to assure that it represents only those states and transitions which are possible in the world it models. Database systems are constrained by integrity constraints defining the legal states of the database, as well as by transition constraints and transaction specifications. Transition constraints and transactions define the legal transitions from state to state. Implementing systems which obey the constraints specified is difficult. The problem is that the constraints may be very complex and involve large amounts of information. For this reason, the cost of checking constraints in a straightforward manner can be prohibitive. Thus, it is imperative that only those constraints which could be violated be checked as efficiently as possible, and then only at points where they could be violated. To do this requires sophisticated analysis.

One way of minimizing the expense of constraint checking is to write transactions which, by their structure, are incapable of violating constraints. For example, a transaction which unconditionally deletes a record with a given key value and then inserts a record with the same key value cannot violate the constraint that the key be unique. If executing such transactions is the only method of modifying the database, then constraint checks that are not explicit parts of a transaction need not be made. Deciding that a transaction is in this class and determining the modifications (e. g., incorporation of constraint checks) which must be made to a transaction to put it efficiently into this class are problems which must be solved by a database system development tool.

In order to solve these problems, it is necessary to add an explicit level of formal specification and analysis to the current database system development paradigm of using a semantic data model, schemas, and transaction programs. Our integration of theory into database systems development has produced a database version of the specification-based software development paradigm presented in [Balzer et al. 83]. Our development method incorporates techniques which guarantee that all transactions are *safe*, i. e., incapable of violating database integrity constraints [Walker and Salveter 81, Stemple and Sheard 84]. The verification technique can also be used in proving semantic properties of specific transactions, much in the manner of [Kemmerer 84] but in a more purely database context, and can also be used to find redundant tests in transactions. Other features of our development method include rapid prototyping and system-generated advice on internal design. We will not discuss the latter feature in this paper.

The general system development method is illustrated in figure 1. A designer starts by describing the real world entities and relationships using some *semantic data model* such as the entity-relationship model [Chen 76] or the semantic hierarchy model [Smith and Smith 80]. This level of specification is translated into a specification of information structures and transitions, i. e., a schema and transactions, with the possible addition of constraints not specifiable in the semantic model. This specification is analyzed by an inference mechanism, e. g., a theorem prover, the results being used along with implementation choices, e. g., file structures, to generate optimized implementations in some programming language and database management system. The information level specification should be executable as a prototype prior to the system's implementation in order to aid the designer in understanding the behavior of the specified system. The observable behavior of the prototype and the results of the formal analysis are valuable feedback to the designer on the quality of both schema and transaction designs.

In figure 2, we give the development paradigm in terms of our choices for an information level language, ADABTPL (to be illustrated later in the paper), and an inference mechanism, a Boyer-Moore style theorem prover [Boyer and Moore 79]. The form of the axioms and functions is the Lisp-like language used by Boyer-Moore. This permits us to use a Lisp interpreter as our prototype interpreter. The *safety theorems* in figure 2 are theorems stating that transactions obey the database integrity constraints. Proofs of these theorems can be used to avoid run-time checks of constraints; this trades compile-time analysis effort for the expense of run-time checks. Inability to prove a safety theorem may indicate that either the transaction or the integrity constraints are faulty. Three alternative actions can be taken in this case: The system can transform the transaction into a safe version; the designer can rewrite the transaction; or the designer can adjust the integrity constraints.

The remainder of the paper illustrates the information level specifications and the formal underpinnings of the method. We first give a schema for an example adapted from [Gerhart 83] and show the axioms which the system generates from the type definitions of the schema. We then consider a transaction and discuss its translation into recursive functions and its analysis. We then briefly outline the manner in which the system supports rapid prototyping. Finally we discuss the building of the database theory which the theorem prover uses as the "knowledge base" it needs for its work.

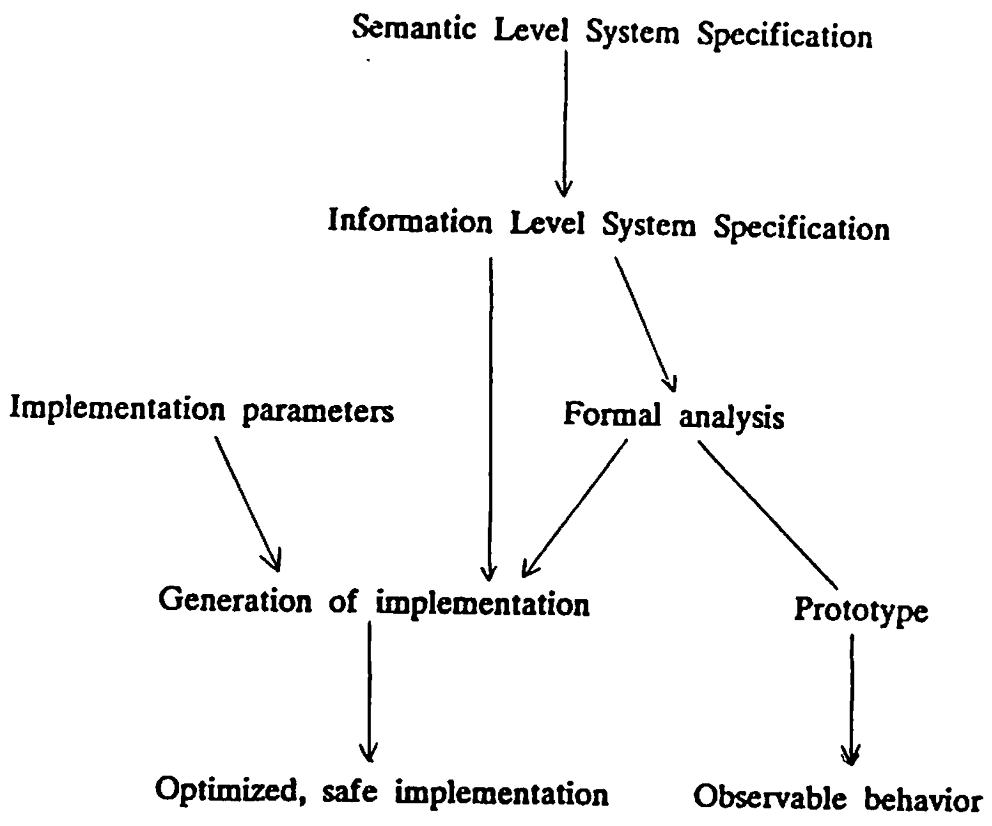
Semantic Level System Specification
Information Level System Specification
Implementation parameters
Formal analysis
Generation of implementation
Prototype
Optimized, safe implementation
Observable behavior

Figure 1: Database System Development Model.

2. Database Specification: Structure and Integrity Constraints

A database schema for a particular database system in our approach consists of a set of type declarations in the Abstract DataBase Transaction Programming Language (ADABTPL, pronounced adaptable). The type declarations build a set of structures, tuples and sets culminating in the definition of a database type whose value set contains all legal databases of the system. In a typical case, ADABTPL is used first to define some tuple types from the set of primitive types, e. g., integers and character strings. Then the tuple types are used as a basis for the finite set

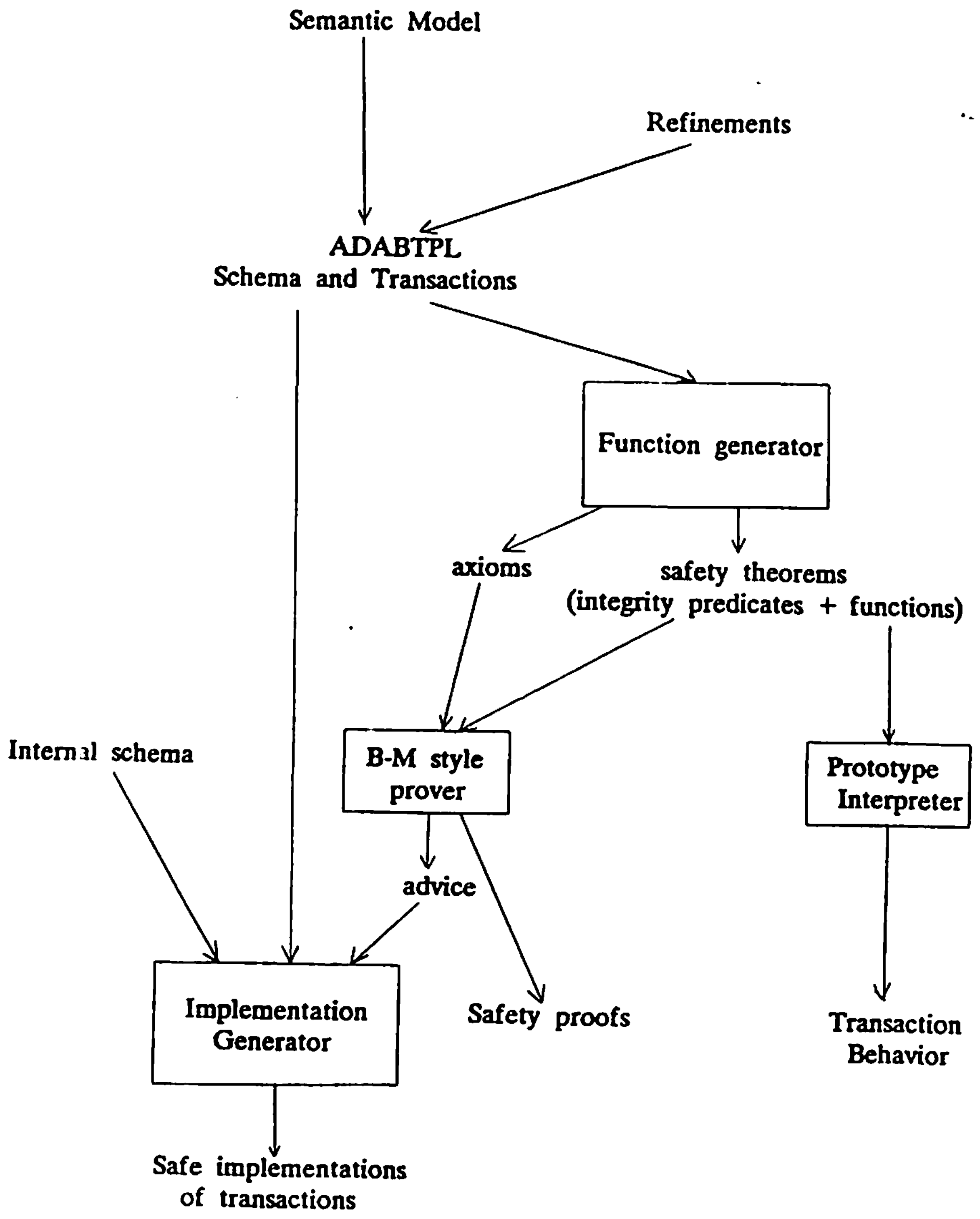

Figure 2: Database System Development using ADABTPL.

types, the relation schemes, of the database. Finally, the database type itself is specified as a tuple type whose constituent types are the relation types. An instance of the database is a tuple, each component of which is a database relation. At each stage of type definition, the designer may introduce predicates defining the constraints on the constituent being defined, domain, tuple, relation, or database. Thus all integrity constraints, including interrelational contraints (a part of the database type declaration), are integrated into the schema in a coherent fashion.

We now present an example to illustrate the ADABTPL type definition language. First we describe the application we will use throughout the paper and give an entity-relationship diagram for it. The application is adapted from the example used in [Gerhart 83]. The database is to be used in managing a job agency. Persons apply for positions, companies subscribe to the service by offering positions, and companies hire and fire employees. Persons who currently do not hold a job are candidates. Only candidates may be hired and only by companies with vacant positions.

Figure 3 is an entity-relationship diagram for a database which will support the job agency application.

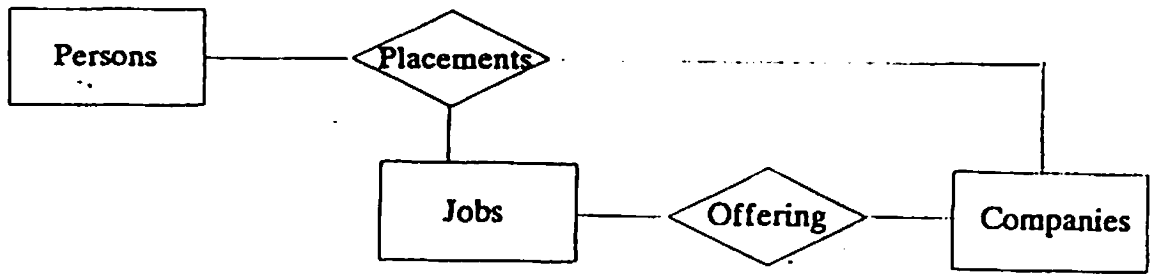

Figure 3: Entity-Relationship Diagram for Job-Agency.

Figure 4 shows the ADABTPL type declarations for a straightforward relational implementation of the Job_Agency E-R diagram. The period (.) is used both for selection of a tuple component and for relational projection, depending on the type of the variable. Projection on two or more components of a tuple or relation V is written V . $[C_1, C_2, \ldots C_n]$.

Note that key constraints in figure 4 are in relation type declarations which can be separate from tuple definitions (e. g., the declaration for Offer). Interrelational constraints can be included in the database type declaration, in this example to express referential integrity and the non-totality of certain of the relationships. It is these constraints and the constraints of the component types which may not be violated by the transactions.

3. Abstract Data Type Axioms from the Database Schema

Each type definition in ADABTPL supplies parameters to a generic abstract data type definition and causes the generation of a set of specific axioms for operations on the database component being defined. These components are either tuples or finite sets. (Lists are also a feature of our system, but will not be discussed in this paper.) The axioms have been designed to capture the semantics of operations on the database in a manner which supports the use of the Boyer-Moore approach in proving transaction safety and other properties of the system [Boyer and Moore 79, Stemple and Sheard 83].

```
Schema of Job-agency

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

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

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

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

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

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

Job_agency = [Persons: Personrel, Jobs: Jobrel, Offering: Offerrel,
              Companies: Companyrel, Placements: Placementrel]

 where Persons . Pid Contains Placements . Pid and
       Companies . Cid Contains Placements . Cid and
       Jobs . Jid Contains Placements . Jid and
       {Placement is a ternary, non-total relationship.}

       Jobs . Jid Contains Offering . Jid and
       {The offered jobs must be known jobs, but
        not all jobs are necessarily offered.}

       Companies . Cid Contains Offering . Cid and
       {All companies offering jobs are known companies,
        but not all companies offer jobs at all times.}

       For all P In Persons: If P . Pid In Placements . Pid
                                   Then P . Placed = 'yes' Else P . Placed = 'no'
       {If a person has a Placement, Placed status is 'yes', else 'no'.}.
```

Figure 4: ADABTPL Schema for Job-Agency Database.

Each tuple type definition leads to the axiomatization of a tuple con.tructor operation (named by the type name) and of a set of selector operations (named by the component names), one for each component of a tuple. The axioms specify the expected reciprocity of construction and selection. For example, the definition of the Offer tuple in figure 4 leads to the axioms in figure 5.

```
Cid(Offer(c, j, n, s)) = c
Jid(Offer(c, j, n, s)) = j
Number_of_positions(Offer(c, j, n, s)) = n
Comments(Offer(c, j, n, s)) = s
```

Figure 5: Axioms for Tuple Type Offer.

The first axiom in figure 5 states that constructing an Offer tuple from c, j, n, and s, written Offer(c, j, n, s), and then selecting a part using the Cid selector function returns a result equal to c. Such axioms are produced by the system and used by the theorem prover. They are rarely to be read or used in any way by humans.

The finite set axioms used to axiomatize relations define an abstract data type with operations *insert, choose, rest*, and *empty* and are given in Figure 6. The choose and rest operation provide the novelty in our treatment of finite sets as an abstract data type. The axioms are *sufficiently complete* in the sense of [Guttag 80] and *safe* in the sense of [Phillips 84]. More importantly, they have been effective in providing the basis for proofs of our database theorems using Boyer-Moore techniques. We have mechanically proven over three hundred theorems and have yet to fail in proving a theorem we have tried, though many have required the proof of several lemmas. (We have tried to prove many non-theorems and have failed every time.)

The generation of a set of axioms for finite sets might not seem to be a significant problem, given the number of axiomatizations of sets in the literature. However, our requirements included two features which were not present together in any of the explicit axiomatizations we could find. The two features we required were: a function which maps from sets to set elements (to allow us to range through the tuples of a relation, for example), such as *Randomtuple* **[Schmidt 77, Casanova et al. 84]**, or *choose* **[Guttag 80, Furtado and Veloso 81]**; and support for structural induction (induction on the structure of a data domain **[Burstall 69]**) as a proof rule. The second feature is a goal of many axiomatizations of abstract data types, but is most often achieved by requiring that the last element used in constructing an instance of a type be immediately selectable by a selector function **[e. g., Standish 78, Boyer and Moore 79, Oppen 78]**. This LIFO behavior is appropriate for stacks, lists, binary trees, and many other data types, but not for sets (nor for that matter, queues). Most treatments of set types either avoid a function which returns an element of a set, using instead an *element-of* and *delete*, or rely on a total order on the element type. We follow the second of these courses, but hide the order in such a way as to make the finite set type exhibit just those properties of sets which are appropriate and still allow the use of structural induction.

The choose operation returns an arbitrary member of its finite set argument. The member is arbitrary but is the same for equal sets. This operation induces an order on the elements of a finite set, but does not impose any particular order. However, the axioms require that any valid implementation for choose not rely on the order of insertion for the choose order. This feature of the axioms constitutes the major difference between finite sets and the list abstract data type with operations CAR, CDR, and CONS. Though it is possible to simulate finite sets using lists, it adds complexities to the proofs of properties needed in analysing database transactions and integrity constraints. See [**Stemple and Sheard 83**] for a further discussion of the ADABTPL axioms and their relationship to lists.

There are three restrictions on the assignment of sorts to the sort variable elements in figure 6. The first is that an equality relation be defined on the sort. The second is that, though elements may be a set type, it may not be the set being defined or be based indirectly on it. Third, the sort substituted for elements must have a *before* function which obeys the axioms involving *before*. An equality relation is the sole requirement for using a type as a tuple component. Since relational equality is defined by the finite set axioms and an order on the relations of a type can be defined in terms of the element order, nothing in our theory restricts relations to first normal form.

Syntax (signature)

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

Basic axioms

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

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

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

insert(e, s) ≠ emptyset
```

Order axioms (normally hidden)

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

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

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

smaller is a *well-founded relation* and
s ≠ emptyset -> smaller(rest(s), s)
(These two axioms restrict sets to finite sets.)

Figure 6: Finite Set Axioms.

4. Transaction Programs: Abstract Database Type Operations

Transactions are the operations of the database system considered as a single abstract data type (the database being the object of the type) and are written in the procedural part of the ADABTPL language. The database type definition, i. e., the type definitions of the database schema, together with type definitions of the input parameters (also in ADABTPL) constitute the declarative part of an ADABTPL transaction. An ADABTPL transaction is composed from update statements on database components, control statements, assignment statements, output statements having no effect on the database, and a header statement declaring the input types and preconditions of the transaction. Though an ADABTPL transaction resembles a typical iterative, non-applicative program such as might be written in Pascal or Pascal R [Schmidt 77], it differs from such programs in three major respects. First, there are no true updatable variables other than the loop controllers. Second, all operations in an ADABTPL transaction are fully axiomatized by the database type definition. Third, ADABTPL transactions have semantics which are expressible as (in fact, defined as) pure recursive functions on the database type. The second and third properties allow us to call the programs abstract specifications and are also the reasons that safety theorems may be proven using Boyer-Moore techniques.

Figure 7 shows the ADABTPL transaction which updates the Job_Agency database when a company hires an employee.

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

Preconditions

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

 Begin {Hire body}

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

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

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

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

End {Hire transaction}.
```

Figure 7: ADABTPL Transaction Hire.

5. Functional Forms of the Transactions

In this section we discuss the translation of ADABTPL programs into recursive functions. The functions are expressed in essentially the language used by Boyer and Moore to express functions and predicates. We call this language the Functional Abstraction Specification Language (FASL, pronounced facile). Transactions written in ADABTPL must be translated into a recursive function form for processing by the theorem prover. The reason for having two languages, one for humans and one for the mechanical prover, will become obvious as we examine the recursive function forms of the transactions. Though this form facilitates the use of Boyer-Moore proof techniques, humans should never be required to write anything complicated in such a language. Designers do not write in FASL. FASL axioms and functions are generated automatically from ADABTPL type definitions and transaction programs, respectively.

Figure 8 gives the FASL form of the Hire transaction given in figure 7 in ADABTPL. Its general form is that of an if statement whose condition is the conjunction of the transaction's preconditions. If the preconditions are false then the original database (db) is returned as the value of the function. If the preconditions are true the function specifies the construction of a new database, by the construction function job-agency. The job_agency function constructs a new database state from the unchanged jobs and companies relations, persons changed by the function fed-person, offerings changed by fed-offer, and placements with a new tuple inserted. The functions fed-person and fed-offer use a generic update function to compute the new persons and offering relations. This function models "for the in a set" statements and is a bit complicated. It takes a relation (passed for technical reasons as two parameters, r and s), two predicates, p and q, a tuple update function, f, and some optional parameters, &x. It returns a relation containing the tuples of r which do not obey p, either minus the tuple of r which obeys p and q, or plus the update by f of the tuple which obeys p but not q. This function is just one of the patterns of updates which occurs commonly and is used only in cases where at most one tuple obeys the predicate p, for example where p specifies a unique key value. There are corresponding update functions for other patterns such as changing all tuples in a relation that obey some predicate. The reason for using such generic functions is that theorems can be proven about them and functions such as fed-person and fed-offer inherit the theorems. This facilitates mechanical proofs of the properties of transactions like Hire.

The functions after update in figure 8 are used to parameterize update to model the "for the" loops in the Hire program. The function falsehood returns false for all input which means that no tuples of persons will be deleted by fed-person.

6. Theorems about Safety and other Properties

A safety theorem for a transaction states that if the transaction is applied to a valid database with valid input and the preconditions are met, then it returns a valid database. Safety theorems allow an implementation to avoid checking constraints not explicitly checked in the body of a transaction, relying on the system to guarantee no incomplete execution of a transaction i. e., guarantee the atomicity of transactions by concurrency control mechanisms. A simple example of a safe transaction that is easily proven safe is one which first deletes a tuple with a given key, say an employee tuple with social security number as key, and then inserts a tuple with the same key. This transaction respects the key constraint, but would lead to a redundant check because of the insert if the delete was not taken into account. This is a very simple example and the general problem of detecting that a transaction is safe is in the most general case undecideable. Even in the case of ADABTPL schemas and transactions, the theorem prover will not be able to prove all safe transactions safe.

```
hire(company, hiree, job, db) =

if (member(tuple($clist(pid), hiree), project(persons(db), pid)) and
   member (tuple($clist(cid, jid), company, job), project(offerings(db), cid, jid)) and
   for-all (persons(db), 'unemployed, hiree)))
   then
     job-agency (fed-person (persons(db), persons(db), hiree),
                 jobs(db),
                 fed-offer (offerings(db), offerings(db), company, job)
                 companies(db),
                 insert (placement(hiree, job, company), placements(db)))
   else
     db))

where

fed-person(r, ans, h) =
  update(r, ans, 'named, 'falsehood, 'mark-employed, h)

fed-offer(r, ans, c, j) =
  update(r, ans, 'comp-job-match, 'last-job, 'dec-num-pos, c, j)

update(r, s, p, q, f, &x)
  if empty(r) then s
     else
       if p(choose(r), &x)
          then if q(choose(r), &x)
                  then delete((choose(r), s)
                  else insert(f(choose(r), &x),
                                   delete(choose(r), s))
          else update(rest(r), s, p, q, f, &x)

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

mark-employed(x, dummy) = person(pid(x), pname(x), paddress(x), 'yes')

comp-job-match(x, c, j) = equal(projT(x, $clist(cid, jid)), tuple($clist(cid, jid), c, j))

last-job(x, dummy1, dummy2) = equal(number-of-positions(x), 1)

dec-num-pos(x, dummy1, dummy2) =offer(cid(x), jid(x), sub1(number-of-postions(x)))
```

Figure 8: FASL Form of the Hire Transaction.

Once a schema has been entered, the designer may write a transaction in ADABTPL and submit it to the system for analysis. The system forms a safety theorem from the FASL translations of the schema and the transaction and submits it to the theorem prover for proof. If the theorem can be proven the designer may be sure that the transaction is sound in terms of obeying the database constraints. If the theorem cannot be proven, the designer may want to examine the transaction or those constraints which the theorem prover cannot verify that the transaction respects as written. If the problem is simply a matter of a missing constraint test, the designer can allow the system to generate a run-time test. Otherwise, the designer can change the transaction or constraint(s) to bring them into conformity.

The safety theorem for the Hire transaction can be expressed in FASL as

```
Consistent(db) and Valid(Company, Hiree, Job, db) ->
          Consistent(Hire(Company, Hiree, Job, db))
```

where Consistent is the predicate function combining the where clauses in the database type definition into a single consistency predicate, and Valid is a predicate on the input values and the database derived from the preconditions and the type definitions of the input (the latter are trivial in this case, but in general, are more complex).

To illustrate a check that can be eliminated in the Hire transaction, consider the statement

```
        Insert([Hiree, Job, Company], Placements)
```

The system must insure that, after the insertion, Placements obeys the four database constraints on it, those in the where clause in the definition of the Job-Agency tuple in figure 4. This could be done by checking each set (say implemented as a file) separately at run-time. It can, however, easily be seen that this need not be done for the inclusion of Hiree in Persons.Pid, since it is a precondition of the transaction and will have been checked if the insertion statement is reached. But what of Company being a known company, and Job being a known job? The preconditions do not state these constraints. Should the system generate the code for the run-time check? No, the validity of the insertion can be inferred from the precondition requiring that [Company, Job] be in the [Cid, Jid] projection of the Offering set. This follows from the constraints that make the Offering job identifier identify current jobs and the Offering company identifier identify current companies. Thus a safety proof of Hire will allow the system to execute the Insert without additional checking.

The requirement that the Placed status of person be 'yes' if the person's identifier is in a Placements tuple is also verifiably obeyed by this transaction and thus need not be checked. Note that this constraint is not met between setting the Placed status 'yes' and inserting the placement tuple. It is just such cases which make considerations of transaction safety more than just optimization and show that treating only simple updates is ineffective and sometimes meaningless. A check after the update of the Placed status would show the database to be inconsistent and always cause the transaction to abort. Questions of which constraints to check and where to check them are very difficult to answer reasonably (all constraints after every transaction is not reasonable) without a powerful inference mechanism and without a sufficiently formal capture of the semantics of transactions. We have proven mechanically the safety of this transaction using our version of the Boyer-Moore theorem prover.

The system can go further than just avoiding the generation of needless checks and actually delete explicitly specified, but redundant, checks in some cases. Suppose the designer had written, instead of the simple Insert,

```
If Job In Jobs.Jid and Company In Companies.Cid
    then Insert ([Hiree, Job, Company], Placements)
    else Reject_transaction ('Bad Placement')
```

We can see from the discussion above that this will always execute the Insert and thus is more expensive than need be.

Write statements must be handled specially since they are not expressible as functional actions on the database, as are all other semantics of ADABTPL programs. In order to correct this, the verification subsystem automatically appends write sets to the database type when asked to prove properties of transaction output. Writes are treated as inserts into their particular write sets. (In some cases, such as when order of writing is semantically important, we use write lists instead of sets, but we will not discuss these cases beyond commenting that they present no more difficulty than do sets.) Assertions about output can then be verified in the same manner as safety is verified.

Theorems are proven using the Boyer-Moore theorem proving techniques. This technique has been shown to be powerful in proving theorems about recursive list functions. The formality brought to database systems by our approach has been designed to produce theorems which closely resemble theorems about list functions, though we have assiduously avoided modelling finite sets as lists. The form of our theorems can be contrasted with those of Gardarin and Melkanoff who use proof techniques based on the Hoare axiomatic approach applied to predicate calculus axioms and assertions imbedded in extended ALGOL 60 transaction programs

[Gardarin and Melkanoff 79]. We believe that the translation of ADABTPL programs into recursive functions will ease the difficulty of proving theorems by stereotyping the theorems, thus making a tailored set of heuristics and lemmas more generally effective than might be the case if functions, axioms, and theorems were "hand-generated". We are currently trying to validate this belief through experimentation.

7. FASL Functions as the System Prototype

Pre-implementation testing can be performed during the system design stage by executing the FASL functions. The FASL functions are LISP-like, and a LISP implementation of the tuple constructor and selector functions, the finite set operations, and the if function is all that is needed to allow the execution of FASL functions directly in LISP.

There are only two non-trivial considerations in these implementations. First the if function cannot behave like an ordinary function in LISP, in that it cannot blindly evaluate all three of its parameters. It must first evaluate its predicate, and then evaluate either the true or false branch depending upon the result. Fortunately most LISP implementations have special constructs for defining such functions.

Second, any implementation of the constructor, selector and set operations must not only implement our intuitive notion of their purpose, but they must also obey the axioms in figure 6. These axioms are not particularly difficult to satisfy, since we designed them first to axiomatize finite sets, and second to describe the way real databases are usually processed (at least abstractly), and we can easily simulate certain file processing.

For example, consider a relation with a key constraint, this could be implemented in a real database using the behavior of a B-tree, where tuples are inserted into their position according to key-order. The functions insert, choose, and delete could be implemented in a similar manner using LISP functions which maintain the tuples in their key-order. We let each tuple be implemented as a list and each relation as a list of tuples, and we define insert, rest, delete, choose, and emptyset by

```
(emptyset) = nil

(insert x r) =
(if (eq r nil)
     (cons x nil)
     (if (before x (car r))
          (cons x r)
          (cons (car r) (insert x (cdr r)))))

(choose r) = (car r)

(delete x r) =
(if (eq r nil)
     nil
     (if (equal x (car r))
          (cdr r)
          (cons (car r) (delete x (cdr r)))))

(rest r) = (delete (choose r) r)
```

The before function tests if its first argument comes before its second in key-order.

Note that the above implementation satisfies all the axioms (we have proven this by using the list axioms [Boyer and Moore 79] and the function definitions to prove the finite set and tuple axioms as theorems) and maintains sets as lists in key-order. We realize that such an implementation is extremely inefficient, but the purpose of the prototype is to convince the designer that the specification specifies what he or she thought it did. Since this purpose is served by relations with few tuples, the benefits obtained from running such a prototype far outweigh the costs involved in executing it.

8. The Extended Theory: The Knowledge Base of the Prover

In order to make the theorem prover effective in dealing with theorems about transactions on highly constrained databases it has been necessary to build a *knowledge base* of database theory. There are two aspects to this knowledge base, the extensions to the theory (function definitions) and the theorems proven and collected in a database accessible to the prover. Choices of both are crucial to the effectiveness of a theoretical design aid. The choice of extensions is partially based on the kinds of constraints that a designer needs to write. For example, key constraints and set containment as well as quantification predicates are essential components of a database constraint language. Therefore these concepts must be "understood" by the theorem prover. To properly deal with the effects of updates on constraints expressed using these concepts, the theorem prover relies on lemmas which capture the interaction behavior of various updates and these constraints. Likewise the concepts of relational algebra should be known to the theorem prover. In figure 9 we give some examples of extensions to the basic theory in terms of function definitions. Figure 10 gives some theorems which we have proven using the theorem prover and which are stored in its lemma list for use in proving transaction theorems.

The definition of a relational update function which replaces all those tuples of a relation r for which p is true with f of those tuples is given in figure 11. This is similar to the update function used in the hire transaction function, but is simplified

```
member(t, R) = if R = emptyset
                 then false
                 else if t = choose(R)
                        then true
                        else member(t, rest(R))

delete(t, R) = if R = emptyset
                 then emptyset
                 else if t = choose(R)
                        then rest(R)
                        else insert(choose(R), delete(t, rest(R)))

project(R, clist) = if empty(R)
                      then emptyset
                      else insert(projT(choose(R), clist),
                                  project(rest(R), clist)

select(R, P) = if empty(R)
                 then emptyset
                 else if P(choose(R))
                        then insert(choose(R), select(rest(R), P))
                        else select(rest(R), P)


key(R, clist) = if empty(R) then true
                  else if member(projT(choose(R), clist), project(rest(R), clist)
                         then false
                         else key(rest(R), clist)

contains(X, Y) = if empty(Y) then true
                   else if member(choose(Y), X)
                          then contains(X, rest(Y))
                          else false

forall(R, P) = if empty(R) then true
                      else if P(choose(R))
                             then forall(rest(R), P)
                             else false

forsome(R, P) = if empty(R) then false
                       else if P(choose(R))
                              then true
                              else forsome(rest(R), P)
```

Figure 9: Extensions of the Basic Theory.

Theorems about select:

member(e, select(R, P)) -> member(e, R)

member(e, select(R, P)) -> P(e)

A theorem about key invariance under certain inserts is

key(R, k) and projT(t, k) ∉ project(R, k)) -> key(insert(t, R), k)

Theorems about contains

X = Y <-> contains(X, Y) and contains(Y, X)

(contains(X, Y) and contains(Y, Z)) -> contains(X, Z)

(contains(X, Y) -> contains(insert(a, X), Y)

Theorems about quantification

forall(R, P) -> forall(delete(e, R), P)

forall(R, P) and contains(R, S) -> forall(S, P)

forall (R, P) and P(e) -> forall(insert(e,R), P)

forsome(R, P) -> forsome(insert(e, R))

forsome(R, P) and contains(S,R) -> forsome(S, P)

forsome(R, P) and not P(e) -> forsome(delete(e, R), P)

Figure 10: Useful Theorems.

```
update-all(r, p, f) =
  if empty(r) then emptyset
              else if p(choose(r))
                   then insert(f(choose(r)), update-all(rest(r), p, f))
                   else insert(choose(r), update-all(rest(r), p, f))
```

Figure 11: A Simplified Update Function.

for purposes of illustration. The predicate *invariant* shown in figure 12 states that if q is true for an element of r then p is true for the function f applied to the element. We have proven

forall(r, p) and invariant(r, q, p, f) -> forall(update-all(r, q, f), p)

This states that if the predicate p is true for all tuples of r and if q is true for a tuple then p is true for f of the tuple, then updating the tuples of r which satisfy q by applying the function f will not violate the constraint that p be true for all tuples in r. The importance of such theorems is their use in avoiding constraint checks. For example, given an update function f and the predicates p and q, checking the forall predicate at run-time can be avoided by a compile-time proof of

p(t) and q(t) -> p(f(t))

which is easily proven in many common cases. This example is illustrative of the sort of "usable" theory which drives the theorem prover in its analysis of database transactions.

```
invariant(r, q, p, f) =
        if empty(r) then true
                else if   q(choose(r)) -> p(f(choose r))
                            then invariant((rest r), q, p, f)
                            else false
```

Figure 12: An Invariant Predicate.

9. Summary

We have presented an approach to specification, verification, and pre-implementation testing of database systems in which a system designer writes a schema and transaction programs in a more or less traditional manner. The system specification produced in this fashion is transformed into formal definitions of abstract data types for the elements of the database and for the system as a single abstract type. The formalization supports optimization of system implementations by allowing needless constraint checking to be avoided. It also supports verification of semantic properties of the system beyond transaction safety, as well as rapid prototyping to test system semantics.

The system described in this paper is currently under development. A version of the Boyer-Moore theorem prover is operational and has been used to prove over two hundred theorems including the transaction safety theorem for the transaction shown in Figure 7, and others of similar complexity. Currently, the translation from ADABTPL into FASL is partially automated. A translator from ADABTPL to FASL and a translator which maps from the entity-relationship model into ADABTPL are under development. Plans include implementations on two commercial database management systems.

References

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

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

[Brodie 81] Brodie, M. L. "Association: A Database Abstraction for Semantic Modeling." In *Entity-Relationship Approach to Information Modeling and Analysis*, P. P. Chen, Ed., 1981.

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

[Casanova et al. 84] Casanova, M. A., Veloso, P. A. S., and Furtado, A. L., "Formal Data Base Specification - An Eclectic Perspective", Proceeding of the Third ACM SIGACT-SIGMOD Symposium on Database Systems, Waterloo, Ontario, April, 1984, pp. 110-118.

[Chen 76] Chen, P. P. "The Entity-Relationship Model - Toward a Unified View of Data." ACM Transactions on Database Systems, Vol. 1, No. 1, March, 1976, pp. 9-36.

[Furtado and Veloso 81] Furtado, A. L. and Veloso, P. A. S. "Procedural Specifications and Implementations for Abstract Data Types." ACM SIGPLAN Notices, Vol. 16, No. 3, March, 1981, pp. 53-62.

[Gardarin and Melkanoff 79] Gardarin, G. and Melkanoff, M., "Proving Consistency of Database Transactions", Proceeding of the 5th International Conf. on Very Large Databases, Rio de Janeiro, Brazil, 1979, pp.291-298.

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

[Guttag 80] Guttag, J. "Notes on Type Abstractions (Version 2)." IEEE Transactions on Software Engineering, Vol 6, No. 1, Jan., 1980, pp. 13-23.

[Kemmerer 84] Kemmerer, R. A. "Testing Formal Specifications to Detect Design Errors." to appear in IEEE Transactions on Software Engineering.

[Lockemann et al. 78] Lockemann, P. C., Mayr, H. C., Weil, W. H. and Wohleber, W. H. "Data Abstractions for Database Systems." ACM Trans. on Database Syst., vol. 4, no. 4, March, 1978, pp. 30-59.

[Oppen 78] Oppen, D. C. "Reasoning about Recursively Defined Data Structures." Fifth ACM Symposium on Priciples of Programming Languages, Tucson, Arizona, Jan., 1978, pp. 151-157.

[Phillips 84] Phillips, N. C. K. "Safe Data Type Specifications." IEEE Transactions on Software Engineering, Vol 10, No. 3, May, 1984, pp. 285-289.

[Santos et al.80] Santos, C. S. dos, Neuhold, E. J. and Furtado, A. L. "A Data Type Approach to the Entity-relationship Model." In *Entity-Relationship Approach to Systems Analysis and Design*, P. P. Chen, Ed., North-Holland, Amsterdam, 1980.

[Scheuermann et al. 80] Scheuermann, P., Schiffner, G., and Weber, H. "Abstraction Capabilities and Invariant Properties Modelling within the Entity-Relationship Model." *In Entity-Relationship Approach to Systems Analysis and Design*, P. P. Chen, Ed., North-Holland, Amsterdam, 1980.

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

[Smith and Smith 80] Smith, J. M. and Smith, D. C. P. "A Data Base Approach to Software Specification" In *Software Development Tools*, Riddle and Fairley, Eds., Springer-Verlag, 1980, pp. 176-204.

[Standish 78] Standish, T. A. "Data Structures - An Axiomatic Approach." in *Current Trends in Programming Methodology, Volume IV, Data Structuring*. R. T. Yeh, Ed., Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Stemple et al. 83] Stemple, D., Ramamritham, K., Vinter, S., and Sheard, T., "Operating System Support for Abstract Database Types." Proceedings Second International Conference on Databases, Cambridge, England, August, 1983.

[Stemple and Sheard 83] Stemple, D. and Sheard, T. "Verifiable Abstract Database Types." University of Massachusetts COINS Technical Report 83-37, November, 1983.

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

[Van Emden and Maibaum 81] Van Emden, M. H. and Maibaum, T. S. E. "Equations Compared with Clauses for Specification of Abstract Data Types." in *Advances in Data Base Theory*, Volume 1, Gallaire, H., Minker, J., Nicolas, J. M., Eds., Plenum, 1981, pp.159-193.

[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).

[Weber 78] Weber, H. "A Software Engineering View of Data Base Systems." Proceedings of the 4th International Conference on Very Large Data Bases, September, 1978, pp. 36-51.

[Yeh and Baker 77] Yeh, R. T., Baker, J. W. "Toward a Design Methodology for DBMS: A Software Engineering Approach." Proceedings of the 3rd International Conference on Very Large Data Bases, October, 1977, pp. 16-27.