Verifiable Abstract Database Types

David Stemple
Tim Sheard

COINS Technical Report 83-37
November 1983

ABSTRACT

A database system, comprising a schema, integrity constraints, transactions, and queries, constitutes a single abstract data type. This type, which we call an *abstract database type*, has as its object the database itself. Thus, the value set of such a type is the set of all legal database states, legal in the sense of obeying all the structural specifications of the schema and the semantic prescriptions of the integrity constraints. The database transactions are the operations of the abstract database type and must be functions on the value set of the type. A transaction specification is *safe* if it defines a function which is closed on the database state set, i. e., any execution of the transaction on a legal database yields a legal database.

We propose an approach to the definition of abstract database types which is both usable by typical database designers and which facilitates the mechanical verification of transaction safety. We present a language, ADABTPL, to be used by database system designers. This language consists of two parts: a type definition part for defining schemas and integrity constraints; and a procedural part for defining database transactions. In order to verify the safety of transactions, axioms on the primitive operations of the system are generated from the ADABTPL type definitions and a pure, recursive function built from the primitive operations is generated from each ADABTPL transaction program. The Boyer and Moore theorem proving technique is then used to prove transaction safety theorems using the recursive functions and axioms.

1. Introduction

A database system, comprising a schema, integrity constraints, transactions, and queries, constitutes a single abstract data type. This type, which we call an *abstract database type*, has as its object the database itself. Thus, the value set of such a type is the set of all legal database states, legal in the sense of obeying all the structural specifications of the schema and the semantic prescriptions of the integrity constraints. The database transactions are the operations of the abstract database type and must be functions on the value set of the type. A transaction specification is *safe* [WALK81] if it defines a function which is closed on the database state set, i. e., any execution of the transaction on a legal database yields a legal database.

This report deals with the specification of abstract database types and the verification of the safety of their transactions. It also touches on the implementation of database systems specified as abstract database types, specifically the avoidance of run-time checks of integrity constraints shown by safety proofs to be unviolatable. We present the preliminaries of a specification language which is usable by database system designers and which is sufficiently formal to allow verification of properties of specifications written in the language. We also discuss the use and enhancement of the Boyer and Moore theorem proving techniques for verifying the safety of abstract database type transactions.

This research has as its goals the development of a technique for defining database systems that is usable by typical database designers and the implementation of a transaction safety verifier which is powerful enough to prove mechanically the safety of most safe transactions specified using the definition technique. The second

goal requires that the definition facility be very formal, and that the proof technique be powerful. We have decided to adapt the Boyer and Moore theorem proving system to the problem of verifying transaction safety since it is very powerful in proving properties of recursive functions on lists. We will demonstrate in this report that the domain of recursive functions on lists has sufficient relevance to the problem of proving the safety of database transactions to warrant investigation of the utility of the Boyer and Moore approach in our work.

1.1 Overview of the Specification and Verification System

Since typical database designers are not likely to be familiar with axiomatic specification, logic, set theory, and proof techniques, the starting point in a database system specification should be something very much like database schemas and transaction programs. Thus, in the proposed system, users begin the specification of an abstract database type by writing type definitions for the components of a database in a form very similar to that of a relational database schema. This schema of type definitions may contain predicates expressing database semantic constraints much like those of [MCLE76] and [BROD80]. However, all constraints, including interrelational constraints, are integrated into type definitions. Transactions are written in a special high-level programming language in a manner similar to the way they are written in current database systems.

The database system specified by the type definitions and transaction programs undergoes two different translations, one into a functional form for the purpose of verification, the other into an implementation in a lower-level language. The first translation transforms the type definitions into a set of axioms and the transactions

into a set of pure, recursive functions. The translation of the transactions into recursive functions is possible since each statement in the transaction programming language is defined as a mapping from one state of the database to another, and the control structures themselves are capable of being translated into recursive functions, though the form of the programming language is neither functional nor recursive.

Attempts are made to verify mechanically the safety of the transactions using their functional forms and the axioms generated from the type definitions. The safety requirement, i. e., the requirement that the range of all transaction functions be the database type, is a major part of the consistency requirement on a system's implementation. The other is the requirement of transaction atomicity which we will not deal with in this proposal. We will assume that transactions are atomic. Since transactions can be complex, their safety requirement requires either run-time checking or proof. A safety proof can be used to obviate the necessity of run-time checks. This follows from the fact that, though individual updates in a transaction may violate constraints, the structure of the transaction and the system's enforcement of transaction atomicity may guarantee the execution of subsequent updates in the transaction which make the constraints true at the end of the transaction (if they were true at the beginning).

The second translation compiles the high-level transactions and type definitions into implementations in a lower-level system, e. g., Pascal R [SCHM77], or even COBOL with Codasyl data manipulation language extensions. It is during this compilation that the transactions are bound to physical files and file access methods. We will call this translation *implementation compiling* and refer to the compiler as the *implementation compiler* to distinguish it from the translator which translates the

transactions into their recursive function form.

The implementation compiler uses results from the verifier's attempt to verify the safety of a transaction. If a transaction is fully verified, the compiler may translate it as is, and the system must thereafter assure only that it runs atomically and serializably using any of the well-studied techniques for achieving this [DATE83]. If, on the other hand, the transaction was not fully verified, the constraints which *may* be violated by the transaction must be checked at run-time, and the compiler must compile these checks based on the information received from the verifier. The capture of the appropriate information by the verifier and its use by the implementation compiler is one of the problems to be solved as this work progresses.

The information that certain constraints may be violated by a transaction could, of course, be of interest to the writer(s) of the transaction and constraints. This information may indicate that either 1. the definition of the transaction is incorrect in the sense that it doesn't do what the designer wanted it to do; 2. the constraints are incorrect in a similar sense; or 3. the transaction, though correct, is not combining enough updates to assure integrity without run-time checking. In any of these three cases, the designer may decide to change either the constraints or the transaction. Thus, the verification phase may be used in the design phase for improving parts of the system design. The manner of presenting the information about verification failure to a designer, who may not be at all familiar with either the axioms generated from the schema or the functional form of the transactions, is another problem being addressed in this work, but is not discussed in this report.

1.2 Relationship to Other Work

The system being described here has goals which overlap those of the work of Gardarin and Melkanoff [GARD79], i. e., the development of a transaction consistency verifier which can be used in the development of safe transactions, but differs in the following respects. We use a type construction approach in which the database constraints are entered as an integral part of automatically axiomatized abstract data type definitions, while in [GARD79], the integrity constraints are expressed separately in predicate calculus. In our approach, transactions are written in a special, high-level language designed specifically for the purpose, not in an ALGOL-like language extended with relational operations and assertions. The most important differences are that the transactions written in our programming language are translated into recursive functions on the database type and that Boyer and Moore techniques of mechanical theorem proving [BOYE79] are used to prove the safety of the transactions in their recursive function form. Gardarin and Melkanoff use proof techniques based on the Hoare axiomatic method [HOAR69] applied to programs in their extended ALGOL 60. Our approach is based on the belief that the human factors of our overall system design in general, and the specification language in particular, are superior to those of the Gardarin and Melkanoff scheme; and on our belief that the proof techniques of Boyer and Moore will prove more safety theorems with less human intervention than a mechanical verifier using a Hoare axiomatic basis.

Our work can also be compared to that of Gerhart in which a simple database application is validated by methods including verification and testing [GERH83]. While there are several differences between this work and ours, including proof techniques, the two major distinctions lie in the treatment of the database, i. e., the state of the system, and in the starting point of the sytem's specification. Using Gerhart's scheme, a user starts with the informal requirements and builds a theory in a typical axiomatic style: axioms on operations and "state variables" are written to capture the essential behavior of the system. In our approach, users start with two predefined, generic abstract data types, tuples and finites sets, and build a type for the state space (the database type) from them. The axioms of the primitive operations on the state vector (database) are not written by users, but are generated automatically from the user-written type definitions designed to be very similar to traditional database schema definitions. These axioms define the operations of the specific tuple and finite set types of the database schema. The definition of an abstract database type constituting the database system is completed by writing transactions, the type's operations, in what looks like a typical, though very high-level, programming language. The transactions are rewritten automatically as recursive functions to facilitate the mechanical verification of their safety and other properties by the Boyer and Moore theorem proving techniques.

In addition to having a different starting point, our approach treats the database differently from Gerhart's method. Gerhart treats the system state in various ways: abstractly and as a sequence of state transitions at her highest level of abstraction, and in increasing degrees of concreteness as she considers levels of implementation. Our database looks like one of her higher-level implementations. This

comparison is validated by the fact that many of her axioms are our theorems, a view which obtains when attempting to verify the correctness of an implementation with respect to a set of axioms (i. e., one attempts to prove the axioms from the properties of the implementation). Thus, Gerhart's approach is somewhat higher-level than ours in that she requires the definition of an abstract data type from "scratch", while we insist on the use of two predefined abstract data types as building blocks of an abstract database type. The advantage of our approach lies in not requiring the user to write axioms, and in allowing the use of database system design methods, such as the entity-relationship model, as a natural prelude to the specification of the system as an abstract database type.

As indicated by the discussion above, our approach to database system design is to be used as an adjunct to other database design methods which focus on the structural design of databases at the conceptual level. For example, having decided on the normalized relations or entity and relationship sets of a particular design, a user would write the type definitions for the relations or sets in our specification language, adding integrity constraints as appropriate. Transactions would then be written using the procedural part of the specification language, and attempts could be made to verify their safety and other properties. These attempts would provide feedback to the designers of both the database and the transactions. Our approach augments other database design methods by providing a formal system in which integrity constraints can be incorporated into schema design in a way which facilitates the analysis of both transactions and schemas. The system brings the design and analysis of transactions to the level of formalism and human factors attained in the design of database schemas using relational normalization and the

entity-relationship model, respectively.

1.3 Organization of Report

In the rest of this report we deal with the techniques used to specify abstract database types and to verify the safety of their transactions. First, we present an overview of the specification system: the type definition language, the transaction programming language, and the functional language. Then we discuss the verification system in which the programming language is first translated into the functional language, deriving in the process, the safety theorems, and then a version of the Boyer and Moore prover is used to prove the safety theorems. The translation of the programming language into the functional language is included in this section since we consider it to be the first phase of the verification process, and it involves choices which may determine the difficulty of safety theorem proofs. We then summarize the research questions we are addressing in this work. A sample safety theorem proof generated by our prototype verifier is given in the appendix.

2. Specification of Abstract Database Types

In this section we present the two parts of an abstract database type specification in our system. The first is the definition of the constituent types of the database type and of the database type itself using a type constructor approach. The second part of the specification defines the operations on the database type as programs in a high-level transaction programming language. First, we give a brief overview of some of the approaches to applying the abstract data type paradigm to database systems.

There have been many formulations of an abstract data type view of database systems [BROD81, EHRI78, GERH83, LOCK78, SANT80, SCHE80, SMIT80, VANE81, WEBE78, YEH77]. Among the goals of these efforts have been good software engineering [YEH77, WEBE78, SMIT80, GERH83], rigorous proofs of system properties [EHRI78, GERH83], and even rapid prototyping for evaluating database system specifications [VANE81, GERH83]. Most researchers have concentrated on the abstract data types for the components of a data base, e. g., relations. Only in [VANE81], and in a sense in [GERH83], do we find the database itself considered to be the object of an abstract data type. While some of these efforts are formal [e. g., EHRI78, GERH83, LOCK78, SANT80, VANE81], none have been developed to the point of mechanically proving the safety of complex operations on the database, with the exception of [GERH83].

We take a type construction approach to the specification of database state constraints, both the structural constraints, i. e., the schema, and the integrity constraints which are traditionally outside the schema. Four type constructors are used, basically equivalent to a set suggested by Brodie [BROD81]. The first is *classification* for the purpose of renaming, typically used in the naming of the constituent types of tuple types, e. g., domain specification for relational tuples. The second is *aggregation* for use in defining tuple types. The third is *grouping* used mainly for defining relation types from tuple types. The fourth is *derivation* which is used for defining types whose value sets are definable as set expressions on other types' values sets. This is used most often to introduce constraints which have the effect of defining subtypes. For example, a key constraint is used to derive from a relation type a subtype containing only those relations which obey the key constraint.

In order for the type definitions constructed using these four constructors to be useful in proving the safety of the functional versions of the transactions, (expressed in terms of the primitive operations of the defined types), each type definition must lead to a set of axioms on the operators of the types. Furthermore, these axioms must be of a form which aids the proof methods adopted. Thus, finite sets as abstract data types must be captured in a manner amenable to the production of induction proofs, since induction is a cornerstone of the Boyer and Moore style of proof.

One of the unique features of the approach taken in this work is the explicit treatment of the database type, i. e., the state specification part of the abstract database type. This is a tuple type in that it is formed by aggregating all the relation types of the database. The state of the database is a single tuple (an instance) of this type. Components of the tuple are the relations (instances of the relation types). Derivation on a database type is the method used to introduce interrelational constraints into the system specification in the same manner as domain constraints in tuple types or relational constraints such as the key property are specified. Transactions which are said to change the state of the database are functions which take a database type tuple (and some input) and return a new database type tuple.

The types of input to a transaction are defined using the same constructors as are used for the database type. This allows the definition of complex input types and the automatic axiomatization of operators on them. One of the benefits of this is that we can treat any transaction as a function with the database and its input types as the domains. The range of any transaction is the database type.

We now define in more detail the four type constructors introduced above, which are used to build up the database type, the type whose value set comprises all legal states of the database.

2.1 Classification

Classification is used to define a type having the same value set as some previously defined type or some primitive type. Operations defined on the predefined or primitive type are defined on the new type as a result of classification.

Examples of the definition of classified types:

Type age = integer

Type name = char20

For primitive types (and for types *derived* from them), e. g., integers and character strings, we assume the existence of an equality relation (=). We also define the predicate *primtypeP* to be true for elements of all the primitive types and false otherwise. We then start the definition of a general equality function *equal* by the partial definition

```
(equal x y) ::= (if (and (primtypeP x) (primtypeP y))
                    (if (= x y)  TRUE  FALSE)
                    ...)
```

We use a Lisp-like language for all functions and predicates. In this notation, *if* is a function of three arguments which returns the second argument if the first is true and the third if the first is false. (In other words, it is the *if-then-else* function.) The ellipsis stands for the definition of equality for other types which we give below.

2.2 Aggregation

Aggregation is the association of several types into a single new type whose value set is the cartesian product of the value sets of its constituent types. We use the term in much the same manner as in [SMITS0]. We will use the following notation. Let K_1, ..., K_n be previously defined types. Let S_1, ..., S_n be n unique names referring to the n components (called attributes) of the new type, and T be the name of the new type. We write

Type T = Aggregation of (S_1: K_1, ..., S_n: K_n)

where no K_i is T, nor is any K_i constructed using T. T is defined by this statement to be an n-ary *constructor* function on the value sets of K_1, ..., K_n. The constructor function and the type have the same name. The function T returns values of type T. In addition to the definition of T, the specification above defines the *destructor* (selector) functions, S_1, ..., S_n, on objects of type T. The destructor functions return values of their corresponding types in the aggregate definition. For example, if X is of type T, then (S_n X) is of type K_n.

Example definition of an aggregate type:

Type Emptuple = Aggregation of (Eno: empid, Ename: name, Sal: saltype)

In addition to named aggregate types, we will often have to deal with anonymous aggregate types. These are the unnamed types of objects created by the projection of other aggregate type objects on a subset of their attributes. We will use the notation (tuple N_1 a_1 N_2 a_2 ... N_n a_n) to mean a call of function *tuple* which creates an instance of an unnamed aggregate type with destructors N_1, N_2, ... ,N_n from the values a_1, a_2, ... ,a_n.

The axioms for the constructor, destructor, and tuple functions are:

For type T = Aggregation of (S_1: K_1, ..., S_n: K_n) and a_i of type K_i, i=1,n

(equal a_i (S_i (T a_1 ... a_i ... a_n)))

(equal a_i (N_i (tuple N_1 a_1 ... N_i a_i ... N_n a_n)))

These axioms simply state that values used to construct a tuple are the values returned by the destructor functions, whether the construction is done by a named constructor (T) or by the constructor of anonymously typed tuples (the tuple function). The axioms are well-defined if the K types are primitive types. In order to extend the equal function to pairs of tuples and to give axioms on a projection function, we introduce the following axioms and definitions.

For a tuple type T, (TP t) is true if and only if t is an element of the value set of the type. TP is the recognizer predicate for type T. We also define the generic tuple recognizer *tupleP* to be true on tuples and false otherwise. For each tuple type T the following is an axiom

(implies (TP t) (tupleP t))

In addition, we define the function *width* into the natural numbers by

(width t) ::= (if (not (tupleP t)) 0 n_T)

where n_T is the number of destructor functions for type T (possibly anonymous).

We can now extend the definition of equal to tuples by

```
(equal t1 t2) ::= (if (and (tupleP t1) (tupleP t2))
                      (if (not (equal (width t1) (width t2)))
                          FALSE
                          (if (and (equal (S11 t1) (S21 t2))
                                   (and (equal (S12 t1) (S22 t2))
                                        (and ...
                                             (equal (S1w t1) (S2w t2))...)
                              TRUE
                              FALSE)
                      ...)
```

where w is the width of the two tuples when they are of the same width, and S_{1i} and S_{2i} are the *ith* destructor functions of t_1 and t_2 respectively, for i=1,w. This defines two tuple to be equal if and only if they have the same width and have equal components, though the selector functions may be different. The ellipsis stands for the part of the definition for comparisands which are not both tuples.

We can now define a projection function *projT* on tuples. If X is an instance of the named aggregate type T described above, with destructors S_1, S_2, ... ,S_n, then (projT S_i ... S_j X) with $1 <= i,...,j <= n$ returns an instance of an anonymous aggregate type with attributes of the same types as S_i, ..., S_j. The following are axioms for projT on named and unnamed tuple types.

```
(equal (projT Si Sj (T a1 ... an))
       (tuple Si ai Sj aj))

(equal (projT Ni Nj (tuple N1 a1 ... Ni ai ... Nj aj ... Nn an))
       (tuple Ni ai Nj aj))
```

2.3 Grouping

Grouping constructs one type from the finite subsets of another. If g is an instance of a type formed by grouping type T, then g is a finite subset of the value set of type T. If T is an aggregate (tuple) type as above, we can define a relation type R by grouping T. We use the keyword *Set* to denote grouping.

Examples of relation types defined using grouping:

Type R = Set of T

Type Employees = Set of Emptuple

Grouping can not define a group type inductively on itself either directly or indirectly.

The following functions are defined on objects of types formed by grouping: *insert*, *delete*, *empty*, *equal*, and *choose*. Of these, only choose deserves discussion. The choose function returns an arbitrary member of its finite set argument. Though the member is arbitrary, it does not vary from one application to another on the same set since choose is a function. The importance of choose, and of its related, defined function *rest* (see below), lie in their use in building recursive functions on finite sets so that inductive proofs can be constructed. We also define the empty set, written *emptyr* since most group types of interest are relation types, to be in the value set of all group types unless removed by derivation.

We define the recognizer function *fsetP* to be true on elements of group types and false otherwise, and include the axiom

(implies (RP r) (fsetP r))

for any R defined as a group type, where RP is the recognizer function for type R.

Before we give the axioms for grouped types, we will define three functions which simplify the statement of the axioms. First we define the function *mem* be the boolean function that indicates membership in a relation. It has the following recursive definition.

```
(mem x y) = (if (empty y)
                false
                ((if (equal (choose y) x)
                     true
                     (mem x (rest y))))
```

We also define the function *rest* as follows:

```
(rest x) = (if (empty x) emptyr (delete (choose x) x))
```

The function contains is defined by

```
(contains x y) = (if (empty y)
                     true
                     (if (mem (choose y) x)
                         (contains x (rest y))
                         false))
```

The following is an initial set of axioms for grouped types which we have found useful during our first efforts to prove safety theorems. We expect that these will evolve as our experience increases. These axioms define an abstract data type for finite sets with operations, insert, delete, empty, and choose.

Equality of finite sets

```
(equal R S) ::= (if (and (fsetP R) (fsetP S))
                    (if (and (contains R S) (contains S R))
                        TRUE
                        FALSE)
                    ...)
```

See below for the complete definition of equality and a discussion of the conditions on its completeness.

Axiom about empty and emptyr

(empty emptyr)

The empty set is empty.

Axioms about insert

(mem x (insert x R))

The element x is in a set after it is inserted.

(implies (mem x R)
(equal (insert x R) R))

If x is in a set, then inserting x does not change the set.

(not (empty (insert x R)))

A set is not empty immediately after an element is inserted.

(implies (mem x R)
(mem x (insert y R)))

If x is in a set, then it is still in the set after y is inserted.

Axioms about delete

(not (mem x (delete x R))))

x isn't in a set after it is deleted.

(implies (not (mem x R))
(equal (delete x R) R))

If x is not in a set, then deleting x does not change the set.

(implies (not (mem x R))
(equal (delete x (insert x R)) R))

If x is not in a set, the set is unchanged by an insert of x followed by a delete of x.

```
(implies (and (mem x R)
              (not (equal x y))
         (mem x (delete y R)))
```

If x is in a set and an element not equal to x is deleted, then x is in the resulting set.

```
(implies (mem x R)
         (equal (insert x (delete x R)) R)
```

If x is in a set, then deleting it and then inserting it does not change the set.

```
(implies (not (equal x y))
         (equal (insert x (delete y R))
                (delete y (insert x R))))
```

If x and y are not equal, then the insert of x and delete of y commute.

Axioms about choose

```
(implies (not (empty R))
         (mem (choose R) R))
```

The choose of a set is in the set.

```
(implies (empty R)
         (equal (choose (insert x R)) x))
```

If a set is empty and x is inserted into it, then the choose of the resulting set is x.

```
(implies (equal R S)
         (equal (choose R) (choose S)))
```

The choose of a set is equal to the choose of an equal set.
(The function choose induces an order, but is not constrained by these axioms to any particular order. However, not all orders will suffice. One order which does not obey this axiom is the order of first insertion. We do not believe that the order requirement is a strong restriction since most candidates for choose implementations will induce an order which is the same for two equal sets and will therefore obey the axioms.)

(or (equal (choose (insert x S)) x)
    (equal (choose (insert x S)) (choose S)))

The order induced on a set by choose is a suborder of the order induced on the set after an element is inserted.

We define the function *card* from finite sets to the natural numbers by

(card S) = (if (empty S) 0 (add1 (card (rest S))))

The relation *less* is a *well-founded-relation* on the natural numbers [BOYE79] and the following is an axiom in our system

(implies (not (empty S)) (less (card (rest S)) (card S)))

This limits our system to finite sets and allows us to use induction on those recursive functions on finite sets which use rest properly in their definitions.

For T and R defined as above, we extend the notion of projection to relations, defining function *projR* on relations, and the following is then true.

(equal (projR S_i ... S_j (insert x R))
    (insert (projT S_i ... S_j x) (projR S_i ... S_j R)))

2.4 Derivation

Let T be any previously defined type, and P be any predicate on objects of type T, then all objects x of type T for which (P x) is true constitute the value set of a new type *derived* from T. Simple derivation is specified by appending a where clause to any of the other forms of definition. More complex forms of derivation involving the set operations of union, intersection and minus are also possible, but will not be discussed here. The predicates associated with derivation are constraints which will need to be verified on the range of the transaction functions.

Example of definitions by derivation:

```
Type Fatcat = Emptuple where Emptuple.Sal > 50000
Type Fatcats = Set of Fatcat where Key(Eno)
```

2.5 Equality

In this section we give the complete definition of the function equal. It is well-defined since by the rules of the type constructors all entities are ultimately composed of primitive types, and because of the following three axioms which state the mutual exclusivity of the generic types.

```
(implies (primtypeP x)
         (not (or (tupleP x) (fsetP x))))

(implies (tupleP x)
         (not (or (primtypeP x) (fsetP x))))

(implies (fsetP x)
         (not (or (primtypeP x) (tupleP x))))
```

The following defines the total function equal for any two entities x and y.

```
(equal x y) ::= (if (and (primtypeP x) (primtypeP y))
                    (if (= x y)  TRUE  FALSE)
                    (if (and (tupleP x) (tupleP y))
                        (if (not (equal (width x) (width y)))
                            FALSE
                            (if (and (equal (Sx1 x) (Sy1 y))
                                     (and (equal (Sx2 x) (Sy2 y))
                                          (and ...
                                               (equal (Sxw x) (Syw y))...)
                                TRUE
                                FALSE)
                        (if (and (fsetp R) (fsetP S))
                            (if (and (contains R S) (contains S R))
                                TRUE
                                FALSE)
                            FALSE)))
```

2.6 The Programming and Functional Specification Languages

In this section we briefly describe the Abstract DAtaBase Transaction Programming Language (ADABTPL). ADABTPL is to be used by transaction designers and programmers for writing transaction programs. ADABTPL is also the language of the type definitions presented above. These definitions serve as the declarations of ADABTPL programs. Transactions written in ADABTPL are mechanically translated into the Functional Abstraction Specification Language (FASL) for purposes of verification. FASL is basically the language used by Boyer and Moore to express theorems as recursive functions and it is the language in which we have presented the axioms. It is the FASL form of the transactions, and the axioms derived from the database and input types, which are used in the verification of the safety of the transactions. The reader is referred to [BOYE79] for a description of this language and to the next section for how it relates to ADABTPL.

There are three basic executable statement types in ADABTPL: Insert, Delete, and Replace. Their respective functions are to insert an element into some set component of the database, delete some element of a set component of the database, and to replace some component of the database. Expressions which produce values for insertion or replacement are written in terms of tuple construction operators "[" and "]", and selector operations which combine attribute names and ".", e. g. Emptuple.Sal. There are also functions and relational operators defined on the primitive types and logical operators for expressing boolean conditions. The language includes four control structures, If-Then-Else, For-Each (in a set), For-To-Do and While-Do.

The following is an example of a transaction and a database type definition written in ADABTPL. The example is adapted from [GARD79].

```
Tuple types

Type Purchase = Aggregation of
                  (Pdate:date, Pitemname:char20, Pquant:int)
Type Sale = Aggregation of
                  (Orderno:int, Sitemname:char20, Squant:int)
Type Itemrecord = Aggregation of (Itemname:char20, Quant:int)

Relation types

Type Buyrel = Set of Purchase
Type Sellrel = Set of Sale where Key(Orderno)
Type Invrel = Set of Itemrecord where Key(Itemname)

Database type

Type Store = Aggregation of
               (Buy:Buyrel, Sell:Sellrel, Inventory:Invrel)

           where
           Contains(Inventory.Itemname, Sell.Sitemname)

           and For each Itemname in Inventory
            Sum(Buy.Pquant
                 where Buy.Pitemname = Itemname)
            - Sum(Sell.Squant
                    where Sell.Sitemname = Itemname)
            = (Inventory.Quant
                 where Inventory.Itemname = Itemname)
```

The two conjuncts in the database type express the constraint that only items in inventory may be sold and that the amount in inventory is the amount bought minus the amount sold.

The following transaction (T3 of [GARD79]) adds a purchase item to inventory.

```
Transaction Add_to_Inv (Newpurchase:Purchase)

  For Each entry In Inventory Do
    Begin
     If Newpurchase.Pitemname = entry.Itemname
        Then
          Begin
            Replace (entry.Quant,
                      entry.Quant + Newpurchase.Pquant)
            Insert (Newpurchase, Buy)
          End
     Endif
   End Add_to_Inv
```

3. Verification of Abstract Database Types

In this section we outline the translation of ADABTPL to FASL, discuss our use of the Boyer and Moore technique of theorem proving, and give the lines of investigation we are proposing in the area of verification. First, we give an overview of formal work in the area of constraint checking.

Except for [GARD79] and [NICO83], formal reasoning about the effect of updates on integrity constraints has concentrated on simple updates. Much of the work has concentrated on simplifying the constraints in order to minimize the amount of data to be accessed and thereby reduce the cost of checking the constraints [TODD77, BERN80, BERN81, BERN82, NICO83]. In other work, efforts are made to insure that only constraints that could possibly be violated are checked [HAMM78, BUNE79, GARD79, BERN80, BERN81]. Our work fits the last pattern, with the difference that we are interested in transactions, not simple updates.

The reason for our interest in transactions, and the limitations of much of the previous work, is that safe transactions may violate constraints during their processing, but are guaranteed by their safety property and the system's maintenance of their atomicity to leave the constraints unviolated when they terminate. Thus, checking the constraints at any point in the progress of a safe transaction is wasteful. Furthermore, some constraints are violated by any of a class of useful and necessary operations, and should never be checked in isolation. For, example, a tuple in a relation containing an item which is the count of tuples in another relation, will be incorrect after either it has been updated or a tuple in the second relation is inserted or deleted. In this case, an atomic combination of two updates is the minimum that should be allowed to execute, and, if atomicity is assured, no checks need be made.

3.1 Translation of ADABTPL into FASL

Each of the operators, statement types, and control structures of ADABTPL has a mapping into FASL. Sequences of statements in ADABTPL map into the nested composition of their FASL functions. This is permissible since each FASL function is defined using the database type as both range and as one of its domains. We now present templates for translating the constructs of ADABTPL into FASL. For ease of reading, most symbols in ADABTPL will begin with capitals and the symbols in FASL, except for constructor and destructor functions, will be completely in small letters. We will use the following type definitions, which are declarations in ADABTPL, for our descriptions.

Type T_i = Aggregation of (D_1: K_1, ..., D_{mi}: K_{mi}) (i = 1,n)

Type RT_i = Set of T_i (i = 1,n)
Type DBT = Aggregation of (R_1: RT_1, ..., R_n: RT_n)

The templates given below are very similar to the partial functions associated with the elementary statements of an ALGOL-like language given in [MANN72]. An ADABTPL construct with a superscript of FASL stands for the FASL translation of the ADABTPL construct. We will use t_i to stand for an instance of type T_i and db to stand for an instance of the database type DBT. S stands for an ADABTPL statement, S_1 S_2 stands for the sequence of statements S_1 and S_2.

$[S_1\ S_2]^{FASL} = (S_2^{FASL}\ (S_1^{FASL}\ db))$

$[Insert(t_i, R_i)]^{FASL} =$

(DBT (R_1 db) ... (insert t_i (R_i db)) ... (R_n db))

The R's select the relations of db. The insert function inserts t_i into the relation (R_i db), and the constructor function DBT puts the new R_i and the other unchanged relations back together into a new database state.

$[Delete(t_i, R_i)]^{FASL} =$

(DBT (R_1 db) ... (delete t_i (R_i db)) ... (R_n db))

$[Replace(R_i, F)]^{FASL} =$

(DBT (R_1 db) ... (R_{i-1} db) (F^{FASL} db) (R_{i+1} db) ... (R_n db))

F is a function (call) on some component of the database, most often R_i, and is translated to a function on the database. Its purpose is to generate the new version of R_i in the new database state.

[If P Then S_1 Else S_2 Endif]^{FASL} =

(if P^{FASL} (S_1^{FASL} db) (S_2^{FASL} db))

In the next three templates ftd, fed, and w stand for families of functions. The particular function generated depends on the actual ADABTPL expressions or statements substituted for exp_1, exp_2 and S in what is treated essentially as a macro call.

[For I = exp_1 To exp_2 Do S(I)]^{FASL} =

(ftd exp_1 exp_2 db) where (ftd init fin dbvar) =

(if (greater init fin) dbvar
(ftd (add1 init) fin (S^{FASL} init dbvar))))

exp_1 and exp_2 are integer valued expressions.

[For Each T In Z Do S(T)]^{FASL} =

(fed Z^{FASL} db) where (fed setvar dbvar) =

(if (empty setvar) dbvar
(S^{FASL} (choose setvar) (fed (rest setvar) dbvar)))

[While P Do S]^{FASL} =

(w db) where (w dbvar) =

(if (not (P^{FASL} dbvar)) dbvar ((w (S^{FASL} dbvar))))

In order to prove things about a while loop we must be able to show that it halts. To do this it must be a theorem that

(implies (P^{FASL} dbvar) (less (m (S^{FASL} dbvar)) (m dbvar)))

where m is some measure function and less is a *well-founded-relation* defined on the range of m [BOYE79]. It is, of course, not always possible to prove this even if it is true.

We now give a FASL translation of the ADABTPL transaction Add_to_Inv presented above. The variable db stands for the database state, i. e., an instance of type Store. The relation names Buy, Sell, and Inventory are FASL functions which return their respective relations when applied to the database. Store is a constructor function which puts together a database state from a Buy relation, a Sell relation, and an Inventory relation. The FASL function for the transaction is built from insert and delete functions on the component relations selected using the relation names. For example, inserting s into the Sell relation of database state db is written in FASL as

(Store (Buy db) (insert s (Sell db)) (Inventory db))

In the following, fed is a function generated from the For-Each loop in the ADABTPL transaction. It is recursive and causes the database used in the construction of the new Store to be written as a recursive call to fed. The FASL version of Add_to_Inv is

```
(fed (Inventory db) db) where
(fed setvar dbvar) =
(if (empty setvar) dbvar
  (if (equal (Pitemname Newpurchase) (Itemname (choose setvar))
      (Store
        (insert Newpurchase (Buy (fed (rest setvar) dbvar)))
        (Sell (fed (rest setvar) dbvar))
        (insert
            (Inventory
               (Itemname (choose setvar))
               (add (Quant (choose setvar)) (Pquant Newpurchase)))
            (delete (choose setvar)
                    (Inventory (fed (rest setvar) dbvar))))

      (fed (rest setvar) dbvar))
```

This has already been simplified from the version produced from the straightforward application of our translation templates by repeated application of the rewrite rules

(equal (Buy (Store b s i) b))

(equal (Sell (Store b s i) s))

(equal (Inventory (Store b s i) i))

The arguments to the Store constructor are the first relation Buy, transformed recursively by the insert on the recursive call to fed; Sell unchanged; and Inventory with the tuple to be replaced first deleted, then inserted with the quantity field updated.

If we let P_{Store} stand for the FASL version of the constraint in the Store database type definition, given above, and PurchaseP stand for the predicate which returns true if its argument is of type Purchase, otherwise false, then the safety theorem for Add_to_Inv is

(implies (and (P_{Store} db) (PurchaseP np)) (P_{Store} (Add_to_Inv np db)))

3.2 Factors Affecting the Use of the Boyer and Moore Scheme

In this section, we first discuss the major difference between our system for safety verification and the Boyer and Moore theorem proving system. We then discuss two issues which will have impact on the effectiveness of using the Boyer and Moore approach. These issues are the choice of FASL recursive functions to represent the ADABTPL control structures and the use of database design methodology in the production of the ADABTPL type definitions. We use the Boyer and Moore approach because of its demonstrated power in proving theorems

of the type which are involved in asserting the safety of transactions. However, it has been necessary to discard some of their mechanism, in particular, shells, and it may be necessary to add new heuristics to handle our theorems better.

3.2.1 Shells, Lists, and Finite Sets

Boyer and Moore have applied their techniques to proving theorems about recursive functions on the domain of *shell objects*. Shell objects are tuples of a named type or *shell*. Shells are types with named constructor and destructor functions like ADABTPL tuple types, but may be defined inductively, unlike ADABTPL tuple types. Shells are named by the object constructor function as in our tuple types and can be defined inductively by allowing the type of a component to be the shell itself. For example, the list type is defined by the shell CONS with destructors CAR and CDR, with no restriction that CONS objects, i. e., lists, not be used as either component in the construction of new lists. Shells are very powerful axiom generating constructs. For example, the *natural numbers* are captured by the shell ADD1 with destructor SUB1 (and a bottom element which has no correlate in our tuple types).

We do not use the shell construct since our need is for finite set types and non-inductively defined tuple types. One result of this is that we do not use any of the reasoning based on *type sets*. This latter fact is not problematic since type correctness is a precondition of our functions and is checkable during the ADABTPL to FASL translation. We could use shells for our tuple types since our tuple types are a subset of the shells definable by the shell definition facilty. The reasons we have not done this are that 1. we want to treat tuple types and finite set types in roughly the same manner, and 2. our tuple types require type restrictors on all

components. These reasons are "soft" reasons and our decision could be reversed in the future if we discover an advantage to using the shell mechanism for tuple types.

Sets, on the other hand, are not capturable by shells because of the automatic inclusion in the shell axioms of an axiom which requires the component selected from a shell object to be the component last used to construct it. This requires an object to "remember" the order of its construction, a property of lists and of the natural numbers, but not of finite sets constructed by insertions or unions. (See below for more details on this subject.) Thus, we do not use shells in dealing with FASL theorems, both because we do not need the power of inductively defined tuples, and because we can not use shells to define finite set types.

In order to define the finite set types of a particular database system axiomatically, we include the explicit group type axioms, given in section 2.3, along with all the tuple constructor and destructor axioms and function definitions generated from the type declarations for the database. One way of looking at our approach in Boyer and Moore terms is to say that we have introduced two new kinds of shells, non-inductive, type-restricted, aggregation shells (a subset of their shells having less power), and group shells for finite sets. However, we use these shells, at the ADABTPL level, to generate axioms at the FASL level, which are explicitly made known to our theorem prover, rather than build knowledge of these kinds of shells into the theorem prover itself.

The differences between lists and finite sets are evident in our group type axioms even though they were designed to minimize the differences between the two in order to exploit the Boyer and Moore techniques. The analogues to CAR, CONS, and CDR are obviously choose, insert, and rest. The axiom for list L,

(IMPLIES (NOT (NULL L))

(EQUAL L (CONS (CAR L) (CDR L))))

has as its FASL counterpart for finite set S,

(implies (not (empty S))

(equal S (insert (choose S) (rest S))))

However, the following list axiom, generated automatically by the CONS shell definition,

(EQUAL A (CAR (CONS A L)))

has no simple counterpart since neither the obvious candidate

(equal a (choose (insert a S)))

nor the possibly appealing

(equal S (insert a (delete a S)))

is an axiom for finite sets. The lack of complete analogy may lead to a need for different heuristics than those which are useful in treating list functions. The development of such heuristics is a goal of our work.

3.2.2 Choices in Forms of FASL Functions

Choices of FASL translations of the ADABTPL control structures different from those given in section 3.1 may lead to theorems which differ in the ease of their proofs. For example, the for-to-do construct can be translated by the template

[For I = exp_1 To exp_2 Do S(I)]^{FASL} =

(ftd exp_1 exp_2 db) where (ftd init fin dbvar) =

(if (less fin init) dbvar (S^{FASL} fin (ftd init (sub1 fin) dbvar))))

The difference between this template and the one presented in section 3.1 is that, in

this version, the final value is counted down and the S function is applied on the way "up" the recursion stack. In the other template the counting and application was performed on the way "down" and the resulting database had been computed upon reaching the base case.

The while construct could be translated by the somewhat cryptic

$[\text{While P Do S}]^{FASL}$ =

(w db) where (w dbvar) =

(if (not (P^{FASL} dbvar)) db (S^{FASL} (w (S^{FASL} dbvar))))

In this version, an implementation nightmare, the database is transformed repeatedly until the base case is reached, if ever, and then it starts over with the pristine database, db, and applies S all over again on the way "up". While this may be an insane way of expressing the performance of a computation on a database from an implementation point of view, it may lead to easier induction since the base case is simpler. (Remember that the FASL expression of programs is solely for the purpose of verification and deriving information needed for the generation or avoidance of run-time checks. The compilation of implementation code is the job of the implementation compiler which starts with the ADABTPL version of the program.) The For-To-Do template also has a simpler base case than the one presented in section 3.1. However, the inductive steps required as a result of these templates may be more difficult to deal with than those generated by the templates in section 3.1. We are currently studying these issues by using different translation templates and comparing the attempts to prove the resulting theorems.

3.2.3 Forms of Theories and Theorems

One problem in axiomatizing abstract data types in programming languages is that each new application tends to require a new "theory", i. e., a set of axioms. Research in database management has led to what may be called "generalized application theories", such as the relational model and the entity-relationship model. Thus, the generation of a database system solution to an application may be quite stereotyped in its "theory" due to the user following one of the database design methods in developing the ADABTPL type definitions. Furthermore, the use of ADABTPL in defining the transactions may also lead to reasonably stereotyped safety theorems to be proved. If the theorems do tend to be stereotyped for either of these reasons, then lemmas developed over the course of verifying the safety of a number of transactions will have a reasonably high probability of being useful in new transaction verification, obviously a desirable feature. The investigation of this possibility is part of future work.

In the appendix we give an example of a schema and a transaction in ADABTPL along with their FASL translations and a system generated trace of a proof that the transaction is safe. This proof was generated by a prototype verifier which occasionally needs direction since its heuristics are still minimal. This direction by the user in no way lessens the validity of the proofs.

4. Summary

We have presented a database system specification technique based on the abstract data type paradigm. To define what we have called an abstract database type a user first writes type definitions culminating in the definition of a database type containing all the structural and integrity constraints on the database. The definitions of the database type and of its constituent types are written in terms of four constructors which build tuple and finite set types from primitive types and integrity constraints. Use of the constructors causes axioms on the primitive operations of a system to be generated. To complete the definition of an abstract database type, its operations, the transactions, are written in a special high-level programming language consisting of operations on the constituent types of the database type and a few control structures. The axioms generated by the type definitions are used to prove the safety of transactions in order to avoid costly run-time checking of the database integrity constraints. To facilitate the safety proofs the transactions are transformed into pure, recursive functions on the database object. This permits us to use the Boyer and Moore theorem proving technique in verifying transaction safety. The major benefit of this approach is that systems are specified formally enough to allow safety proofs to be generated mechanically, thus permitting the implementation to avoid costly integrity constraint checking. This is accomplished by a specification language which is sufficiently similar to traditional schema and programming languages to be easily usable by typical database designers.

We now summarize the research questions to be addressed as our work progresses.

1. What is a good database system design and specification language which is easy to read and write by designers and formal enough to allow proofs of transaction safety to be generated mechanically?

2. How can the Boyer and Moore theorem proving techniques be used and enhanced for proving transaction safety theorems of the form presented in this report?

3. What techniques are needed to compile the language of the first question into implementations?

4. How can a transaction verifier, which deals with transactions translated from a programming language into recursive formulas and algebraic axioms, communicate the reasons for verification failure (or success) to writers and compliers of the language in a useful and meaningful manner?

5. In what way does the substitution of finite sets for lists change the set of useful heuristics of the Boyer and Moore approach to mechanical theorem proving?

6. Which among the different recursive functions that can be used to represent the semantics of control structures lead to the easiest safety theorems to prove?

7. Does the use of database systems design methods, such as the normalization of relations and the entity-relationship model, lead to axioms and transactions which promote the ease of transaction safety theorem proofs?

References

[BERN80] Bernstein, P. A., Blaustein, B. T., Clarke, E. M. "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data." Proceedings of the 6th International Conference on Very Large Databases, 1980. pp. 126-136.

[BERN81] Bernstein, P. A., Blaustein, B. T. "A Simplification Algorithm for Integrity Assertions and Concrete Views." Proceedings 5th International Computer Software and Applications Conference, Chicago, November 1981.

[BERN82] Bernstein, P. A. and Blaustein, B. T. "Fast Methods for Testing Quantified Relational Calculus Expressions." Proceedings of 1982 ACM SIGMOD Conference, pp. 39-50.

[BOYE79] Boyer, R. S. and Moore, J. S. *A Computational Logic*, Academic Press, New York, 1979.

[BROD80] Brodie, M. L. "The Application of Data Types to Database Semantic Integrity." Information Systems 5, pp. 287-296 (1980)

[BROD81] Brodie, M. L. "Association: A Database Abstraction for Semantic Modeling." In *Entity-Relationship Approach to Information Modeling and Analysis*, P. P. Chen, Ed., 1981.

[BUNE79] Buneman, O. P., Clemons, E. K. "Efficiently Monitoring Relational Databases." ACM Transcations on Database Systems vol. 4, No. 3. September 1979.

[DATE83] Date, C. J. *An Introduction to Database Systems. vol. II* Addison-Wesley Publishing Company, Reading, Ma. (1983)

[EHRI78] Ehrig, H., Kreowski, H. J., and Weber, H. "Algebraic Specification Schemes for Data Base Systems." Proceedings of the 4th International Conference on Very Large Data Bases, 1978, pp. 427-444.

[GARD79] Gardarin, G. and Melkanoff, M. "Proving Consistency of Database Transactions." Proceeding of the 5th International Conference on Very Large Databases, 1979, pp. 291-298.

[GERH83] Gerhart, S. "Formal Validation of a Simple Database Application." Proceedings of the Sixteenth Hawaii International Conference on System Sciences, 1983, pp. 102-111.

[HAMM78] Hammer, M. M., Sarin, S. K. "Efficient Monitoring of Database Assertions." Proceedings 1978 ACM SIGMOD International Conference on Management of Data. June 1978.

[HOAR69] Hoare C. A. "An Axiomatic Basis for Computer Programming." Communications of the ACM, vol. 12, no.10, October 1969. pp. 576-580.

[LOCK78] 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.

[MANN72] Manna, Z. and Vuillemin, J. "Fixpoint approach to the theory of computation." Communications of the ACM, vol. 15, no. 7, July, 1972, pp. 528-536.

[MCLE76] Mcleod, D. J. "High Level Expression of Semantic Integrity Specifications in a Relational Data Base." Report No. MIT/LCS/TR-165, Laboratory for Computer Science, Massachusetts Institute of Technology (September 1976)

[NICO83] Nicolas, J. M. "Logic for Improving Integrity Checking in Relational Data Bases." to appear in Acta Informatica.

[SANT80] 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.

[SCHE80] 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.

[SCHM77] 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.

[SMIT80] 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.

[TODD77] Todd, S. J. P. "Automatic Constraint Maintenance and Updating Defined Relations." Proceedings IFIP Congress 1977 (North Holland 1977)

[WALK81] 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).

[VANE81] 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.

[WEBE78] 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.

[YEH77] 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.

5. Appendix -- An Example Safety Proof

In this appendix we present a proof of the safety of a "hire" transaction on a simple employee database. The integrity constraint that is to be verified demands that only dependents whose parents are in the employee relation are allowed in the dependent relation.

Below we give the type definitions for this database followed by an ADABTPL version of the transaction. (The ADABTPL presented here differs from that presented in the body of this report. This version is used because it is simpler to use the verifier rewrite rules as a translator for this form.) We present a system generated trace of the proof of its FASL equivalent, showing that the database state resulting from a "hire" obeys the integrity constraint.

Consider the following ADABTPL specification:

```
type name = string
type age  = integer

type employeetuple  =  aggregation of (en: name ea: age)
type dependenttuple =  aggregation of (dn: name en: name)

type employees  = set of employeetuple
type dependents = set of dependenttuple

type children = set of name

type dbt = aggregation of (emp: employees dep: dependents)
                    where (contains (projR en emp) (projR en dep))
projR

db : dbt


Transaction Hire( en: name, c: children, ea: age )
  Begin
    Insert( [en ea] emp )
    For Each x In c Do
      Insert( [x en] dep )
  End
```

By the semantics of the type constructors, the following (among others) are axioms of the system.

```
(implies nil (equal (emp (dbt *x *y)) *x))

(implies nil (equal (dep (dbt *x *y)) *y))
```

```
(implies nil (equal (projR en (insert *x *y))
                    (insert (projT en *x) (projR en *y))))

(implies nil (equal (projT en (employeetuple *x *y)) (tuple *x)))


(implies nil (equal (projT en (dependenttuple *x *y))
                    (tuple *y)))
```

The following two functions are simplified versions of what the ADABTPL to FASL translator would produce from the transaction above.

```
(hire *en *c *ea *db) =
(dbt
  (insert (employeetuple  *en *ea) (emp *db))
  (addchildren (dep *db) *c *en)))

(addchildren *d *c *en) =
(if (empty *c)
      *d
      (insert (dependenttuple (choose *c) *en)
              (addchildren *d (rest *c) *en))))
```

where "hire" is the FASL form of the transaction in question, "addchildren" is an auxiliary function performing the For-Each loop. It is one fed function generated from the For-Each translation template.

The theorem below states that if the database type constraint is met by the input database, then it is met by the output of hire.

The following is an actual log file of the verifier in action with some of the more obvious and repetitious intermediate forms deleted. In all cases every transformation done by the system is recorded, even if the intermediate result is not shown. At this stage of development the verifier occasionally stops and asks for help in deciding which possible avenues to pursue. The user can then suggest ones he believes to be most advantageous. This in no way invalidates the correctness of the proofs produced.

(original theorem)

```
(implies (contains (projR en (emp $db)) (projR en (dep $db)))
         (contains (projR en (emp (hire $en $c $ea $db)))
                   (projR en (dep (hire $en $c $ea $db)))))
```

(expanding by hire)

```
(implies (contains (projR en (emp $db)) (projR en (dep $db)))
         (contains (projR en
                          (emp
                           (dbt (insert (employeetuple $en $ea) (emp $db))
                                (addchildren (dep $db) $c $en))))
                   (projR en
                          (dep
                           (dbt (insert (employeetuple $en $ea)
                                        (emp $db))
                                (addchildren (dep $db) $c $en))))))


(rewriting by rule (implies nil (equal (dep (dbt *x *y)) *y)))

(rewriting by rule (implies nil (equal (emp (dbt *x *y)) *x)))

(implies (contains (projR en (emp $db)) (projR en (dep $db)))
         (contains (projR en (insert (employeetuple $en $ea) (emp $db)))
                   (projR en (addchildren (dep $db) $c $en))))


(rewriting by rule
           (implies nil
                    (equal (projR en (insert *x *y))
                           (insert (projT en *x) (projR en *y)))))

(implies (contains (projR en (emp $db)) (projR en (dep $db)))
         (contains (insert (projT en (employeetuple $en $ea))
                           (projR en (emp $db)))
                   (projR en (addchildren (dep $db) $c $en))))


(rewriting by rule
           (implies nil (equal (projT en (employeetuple *x *y)) (tuple *x))))

(implies (contains (projR en (emp $db)) (projR en (dep $db)))
         (contains (insert (tuple $en) (projR en (emp $db)))
                   (projR en (addchildren (dep $db) $c $en))))


(proof by induction on addchildren)
(induction base case)

(implies (empty $c)
         (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                  (contains (insert (tuple $en) (projR en (emp $db)))
                            (projR en (addchildren (dep $db) $c $en)))))
```

```
(expanding by addchildren)

(implies (empty $c)
         (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                  (contains (insert (tuple $en) (projR en (emp $db)))
                            (projR en
                                   (if (empty $c)
                                       (dep $db)
                                       (insert (dependenttuple (choose $c)
                                                               $en)
                                               (addchildren (dep $db)
                                                            (rest $c)
                                                            $en)))))))


(rewriting by rule
           (implies nil
                    (equal (projR *a (if *b *c *d))
                           (if *b (projR *a *c) (projR *a *d)))))


(rewriting by rule
           (implies nil
                    (equal (projR en (insert *x *y))
                           (insert (projT en *x) (projR en *y)))))

(rewriting by
           rule
           (implies nil (equal (projT en (dependenttuple *x *y)) (tuple *y))))

(implies (empty $c)
         (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                  (contains (insert (tuple $en) (projR en (emp $db)))
                            (if (empty $c)
                                (projR en (dep $db))
                                (insert (tuple $en)
                                        (projR en
                                               (addchildren (dep $db)
                                                            (rest $c)
                                                            $en)))))))

(to conj-normal form)

(((not (empty $c))
  (not (contains (projR en (emp $db)) (projR en (dep $db))))
  (not (empty $c))
  (contains (insert (tuple $en) (projR en (emp $db))) (projR en (dep $db))))
 ((not (empty $c))
```

```
(not (contains (projR en (emp $db)) (projR en (dep $db))))
(empty $c)
(contains (insert (tuple $en) (projR en (emp $db)))
          (insert (tuple $en)
                  (projR en (addchildren (dep $db) (rest $c) $en))))))
```

(throwing away vacuously true clauses)

```
(((not (empty $c))
  (not (contains (projR en (emp $db)) (projR en (dep $db))))
  (not (empty $c))
  (contains (insert (tuple $en) (projR en (emp $db))) (projR en (dep $db)))))
```

(applied lemmas)

```
((true (by applying lemma)
       (implies (contains *a *b) (contains (insert *x *a) *b))
       to
       (contains (insert (tuple $en) (projR en (emp $db)))
                 (projR en (dep $db)))))
```

(induction step is)

```
(implies (and (not (empty $c))
              (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                       (contains (insert (tuple $en) (projR en (emp $db)))
                                 (projR en
                                        (addchildren (dep $db)
                                                     (rest $c)
                                                     $en)))))
         (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                  (contains (insert (tuple $en) (projR en (emp $db)))
                            (projR en (addchildren (dep $db) $c $en))))))
```

(expanding by addchildren)

```
(implies (and (not (empty $c))
              (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                       (contains (insert (tuple $en) (projR en (emp $db)))
                                 (projR en
                                        (addchildren (dep $db)
                                                     (rest $c)
                                                     $en)))))
         (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                  (contains (insert (tuple $en) (projR en (emp $db)))
```

```
                    (projR en
                        (if (empty $c)
                            (dep $db)
                            (insert (dependenttuple (choose $c)
                                                    $en)
                                    (addchildren (dep $db)
                                                 (rest $c)
                                                 $en)))))))

(rewriting by
        rule
        (implies nil
                (equal (projR *a (if *b *c *d))
                        (if *b (projR *a *c) (projR *a *d)))))

(rewriting by rule
        (implies nil
                (equal (projR en (insert *x *y))
                        (insert (projT en *x) (projR en *y)))))

(rewriting by
        rule
        (implies nil (equal (projT en (dependenttuple *x *y)) (tuple *y))))

(implies (and (not (empty $c))
              (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                       (contains (insert (tuple $en) (projR en (emp $db)))
                                 (projR en
                                        (addchildren (dep $db)
                                                     (rest $c)
                                                     $en)))))
         (implies (contains (projR en (emp $db)) (projR en (dep $db)))
                  (contains (insert (tuple $en) (projR en (emp $db)))
                            (if (empty $c)
                                (projR en (dep $db))
                                (insert (tuple $en)
                                        (projR en
                                               (addchildren (dep $db)
                                                            (rest $c)
```

(to conj-normal form and throwing away vacuously true clauses)

```
(((not
   (contains (insert (tuple $en) (projR en (emp $db)))
             (projR en (addchildren (dep $db) (rest $c) $en))))
  (empty $c)
  (not (contains (projR en (emp $db)) (projR en (dep $db))))
  (empty $c)
  (contains (insert (tuple $en) (projR en (emp $db)))
            (insert (tuple $en)
                    (projR en (addchildren (dep $db) (rest $c) $en))))))
```

(applied lemmas)

```
((true (by applying lemma)
       (implies (contains (insert *a *x) *y)
                (contains (insert *a *x) (insert *a *y)))
       to
       (contains (insert (tuple $en) (projR en (emp $db)))
                 (insert (tuple $en)
                         (projR en (addchildren (dep $db) (rest $c) $en))))))
```

QED